Analyse von Hardwarefehlern in Vermittlungseinheiten digitaler Netze

Frank Balbach

Dissertation

Erlangen - 2000

Analyse von Hardwarefehlern in Vermittlungseinheiten digitaler Netze

Der Technischen Fakultät der
Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR - INGENIEUR

vorgelegt von

Frank Balbach

Erlangen - 2000

Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung:	02.02.2000
Tag der Promotion:	
Dekan:	Prof. Dr. H. Meerkamm
Berichterstatter:	Prof. Dr. M. Dal Cin
	Prof. Dr. U. Herzog

Danksagung:

Ich möchte mich ganz herzlich bei Herrn Prof. Dr. Dal Cin für die Betreuung und die Begutachtung dieser Arbeit, ebenso bei Herrn Prof. Dr. Herzog für die bereitwillige Übernahme des Zweitgutachtens bedanken.

Mein weiterer Dank gilt allen Kollegen, die mich mit Kritik und Anregungen bei dieser Arbeit unterstützt haben. Besonders hervorheben möchte ich Volkmar Sieh, ohne dessen Unterstützung bei VERIFY und ohne dessen erhellende Diskussionsbeiträge diese Dissertation nicht zustande gekommen wäre. Desweiteren gilt ein besonderer Dank meinem „Schatz“ Oliver Tschäche für seine bombigen Einwürfe, Susann Allmaier, die dafür gesorgt hat, daß der Lehrstuhl nicht im Grau-in-Grau versinkt und Stefan Dalibor für seine unermüdliche Suche nach Verbesserungen in der Rechnerumgebung.

Doch all mein Bemühen wäre zum Scheitern verurteilt gewesen, wenn mich nicht meine Frau Martina immer wieder so wundervoll ermutigt hätte, die Arbeit zu Ende zu führen. Für all die Wochenenden, die ich wegen der Fertigstellung ihr und meinem Sohn Nicolas nicht zur Verfügung stand, möchte ich mich im nachhinein entschuldigen. Vielen lieben Dank diesen beiden für Ihren Langmut.

Diese Arbeit widme ich meinen Söhnen
Nicolas und Maximilian.

Inhaltsverzeichnis

KAPITEL

1

Einleitung

Wird in den Geschichtsbüchern der nächsten Jahrhunderte das Ende des 20. Jahrhunderts als der Beginn des Kommunikationszeitalters festgehalten sein? Und wird das gegenwärtige Zeitalter vergleichbare gesellschaftliche Umwälzungen mit sich bringen wie schon die Erfindung des Eisens oder die Industrialisierung? Selbst wem diese Einschätzung zu gewagt scheint, wird anerkennen müssen, daß zumindest die Übertragung von Informationen eine stetig wachsende Bedeutung in unserer Gesellschaft einnimmt. Neben der schon lange vertrauten Technik der analogen Telephonie und Radio- bzw. Fernsehübertragung werden heutzutage Informationen immer häufiger auf digitaler Basis übertragen. Eine beispiellos rasante Entwicklung der digitalen Datenverarbeitung ermöglicht es uns schon jetzt, große Mengen an Informationen in kürzester Zeit rund um den Globus zu schicken. Schon heute wird eine stetig wachsende Zahl von Waren und Dienstleistungen über digitale Datennetze, wie zum Beispiel das Internet, vermarktet und verkauft. Banktransaktionen, Steuerabrechnungen oder der Kauf von Büchern über digitale Datennetze sind heute schon alltäglich. Mit Hilfe von Intranets verwalten große Firmen heutzutage ihre internen Produktions- und Organisationsabläufe. Vernetzte Computer steuern Automobile, den Verkehrsfluß auf unseren Straßen und Schienensystemen und werden sogar in der Leittechnik von Kernkraftwerken eingesetzt.

Allen diesen Beispielen ist gemein, daß neben der Geschwindigkeit der Datenübertragung auch die Sicherheit und Zuverlässigkeit entscheidende Faktoren beim Datenaustausch sind. Um das mutwillige Abhören oder Verfälschen der übertragenen Daten zu verhindern, sind dabei mächtige Verschlüsselungsverfahren im Einsatz. Auch das Problem, daß Daten durch Rauschen auf den Übertragungswegen verfälscht werden können, hat man durch geeignete Verfahren, wie zum Beispiel durch einen CRC (Cyclic Redundancy Check)[ShMF 85], im Griff. Demgegenüber existieren nur relativ wenig Informationen über das Verhalten von Vermittlungseinheiten bei Hardwarefehlern. Angesichts der Tatsache, daß die Daten auf dem Weg vom Sender zum Empfänger im Internet oder in großen Parallelrechnern oft mehrere Dutzend Vermittlungsstationen passieren, müssen möglichst detaillierte Kenntnisse über Hardwarefehler innerhalb dieser sogenannten Switches vorhanden sein, um in diesen komplexen Netzstrukturen Fehler erkennen und behandeln zu können. Die Fehlererkennung und -lokalisierung stellt dabei die Basis für eine spätere Rekonfiguration des Netzwerkes dar.

Dies bedarf jedoch einer möglichst genauen Kenntnis des Fehlverhaltens der Vermittlungseinheiten und der Auswirkungen auf das Kommunikationsprotokoll nach dem Auftreten eines Hardwarefehlers. Diesem Ziel einen Schritt näher zu kommen ist die Grundlage dieser Arbeit.

Die Untersuchung des dynamischen Verhaltens eines Kommunikationsprotokolls ist eine Aufgabe, die entweder an der realen Hardware des Netzwerkes oder mit Hilfe einer Modellierung des Systems erfolgen kann. Für die Untersuchung des Verhaltens im Falle eines Hardwarefehlers innerhalb eines Vermittlungsknotens muß jedoch berücksichtigt werden, daß das Auftreten von Hardwarefehlern in der Regel ein Ereignis mit geringer Wahrscheinlichkeit ist. Es ist also notwendig, künstlich Fehler in das System einzubringen. Frühere Arbeiten im Bereich der Fehlerinjektion haben gezeigt, daß die Genauigkeit der Ergebnisse bei der Injektion von Hardwarefehlern erheblich von dem verwendeten Fehlermodell abhängig ist [SiTB 97a]. Aus diesem Grund wurde in dieser Arbeit ein Fehlermodell gewählt, das sich direkt aus der Hardwarebeschreibung der Vermittlungseinheiten ergibt. Dies beinhaltet auch, daß ausschließlich Hardwarefehler berücksichtigt wurden. Auswirkungen von Fehlern in der Systemsoftware (Firmware) wurden im Rahmen dieser Arbeit nicht untersucht.

Die Auswirkungen von Fehlern in digitalen Vermittlungseinheiten wurden in dieser Arbeit am Beispiel zweier Switches mit jeweils unterschiedlichen paketorientierten Protokollen untersucht. Es handelt sich dabei zum einen um den von INMOS entwickelten STC104 [MaTW 93], dessen Kommunikationsprotokoll eine Flußkontrolle auf den unteren Ebenen der Protokollhierarchie beinhaltet. Ursprünglich war der STC104 ausschließlich als Router für den gleichzeitig entwickelten T9000 Transputer vorgesehen. Nachdem der T9000 jedoch nicht die erwartete Verbreitung am Markt gefunden hat, wurden für den Router STC104 neue Anwendungsgebiete gefunden. Beispielsweise läßt sich der Router dazu verwenden, ATM[1]-Zellen zu vermitteln [MaTW 93].

Im Gegensatz zum Protokoll des STC104 ist beim reinen ATM-Protokoll keine Flußkontrolle auf den unteren Protokollebenen vorgesehen und zusätzlich eine feste Paketgröße vorgegeben. Aus diesem Grund wurde in dieser Arbeit ein rein ATM-basierter Vermittlungsknoten als zweites Untersuchungsobjekt gewählt. Es handelt sich hierbei um einen von Yeh et al. [YeHA87] vorgestellten Knockout-Switch. Obwohl beide Router auf einem paket- bzw. zellorientierten Kommunikationsprotokoll basieren, zeigte sich im Laufe der Untersuchungen, daß gerade die Aspekte der Flußkontrolle und die Art der Paketierung der Daten einen qualitativen Unterschied des Systemverhaltens nach Injektion eines Hardwarefehlers bewirken.

1.1 Ziele dieser Arbeit

Wie eingangs schon beschrieben, hängen die Ergebnisse der Fehlerinjektion wesentlich vom Detailgrad des Fehler- und des Hardwaremodells ab. Die in dieser Arbeit verwendete Methode zur Fehlerinjektion basiert auf einem am Lehrstuhl Informatik III der Universität Erlangen-Nürnberg von Sieh [Sie 98] entwickelten Verfahren. Dabei können mit dem von Sieh entwickelten Werkzeug **VERIFY** (**V**HDL-based **E**valuation of **R**eliability by **I**njecting **F**aults

---

[1] ATM: Asynchronous Transfer Mode, ANSI standardisiertes Kommunikationsprotokoll

efficientl**Y**) Fehler während der Simulation eines mit VHDL[1] beschriebenen Hardwaresystems eingebracht und ausgewertet werden. Neben dem primären Ziel dieser Arbeit, d.h. der Untersuchung von Hardwarefehlern in digitalen Vermittlungseinheiten, ist der Bewertung von VERIFY im praktischen Einsatz ein weiterer Schwerpunkt gewidmet. Dabei steht die Verwendbarkeit von VERIFY in frühen Phasen der Hardwareentwicklung im Vordergrund.

Zur Untersuchung des Verhaltens der Systeme im Fehlerfall wurden neben temporären Hardwarefehlern, d.h. Fehlern mit zeitlich begrenztem Auftreten, auch permanente Fehler injiziert. Dabei wurde neben den Auswirkungen auf das jeweilige Kommunikationsprotokoll auch das direkte Ausfallverhalten der modellierten Switches untersucht. Die Modellierung der beiden Switches wurde dazu auf Gatterebene durchgeführt. Damit war es möglich, die Wahrscheinlichkeit zu ermitteln, mit der ein Fehler auf Gatterebene zu einem Fehlverhalten des Switches führt. Neben dem bekannten Stuck-At-Fehlermodell auf Gatterebene wurde dabei ein weiteres Fehlermodell entwickelt, welches das Übersprechverhalten benachbarter Leitungen auf einem Chip berücksichtigt. Die Auswirkungen dieses erweiterten Fehlermodells wurden dann mit dem Stuck-At-Fehlermodell verglichen.

Desweiteren wurde untersucht, mit welcher Wahrscheinlichkeit sich die Switches nach einem Fehler auf Gatterebene selbständig erholen und wie lange diese Fehler dabei in den Switches verbleiben. Zusammen mit den Auswirkungen auf das Protokoll lassen sich damit Hinweise geben, wie diese Fehler erkannt, lokalisiert und behoben werden können.

1.2 Kapitelübersicht

Im zweiten Kapitel dieser Arbeit wird nach einer kurzen Übersicht zur Methodik der Fehlerinjektion das Verfahren von Sieh, zusammen mit dem von ihm entwickelten Werkzeug VERIFY vorgestellt. Desweiteren werden in diesem Kapitel die beiden Fehlermodelle *Stuck-At* und *Übersprechfehler* beschrieben.

Anschließend werden in Kapitel 3 die beiden Switches (STC104 und ATM-Knockout) detailliert vorgestellt. Neben den beiden zugrundeliegenden Kommunikationsprotokollen wird hierbei auf die Hardware der Switches eingegangen und die Modellierungsaspekte bei der Beschreibung mit VHDL und VERIFY dargelegt. Den Abschluß dieses Kapitels bildet die Vorstellung der beiden Lastmodelle, welche die Switches gemäß des jeweiligen Protokolls mit Daten und weiteren Protokollelementen versorgen.

Kapitel 4 beschäftigt sich dann mit den Fehlerauswirkungen der Fehlerinjektionsmessungen, die an den Modellen vorgenommen wurden. Anhand von mehreren tausend injizierten Fehlern auf Gatterebene wird die Wahrscheinlichkeit ermittelt, mit der ein solcher Fehler im Switch verbleibt und wie hoch die Wahrscheinlichkeit einer Selbstheilung des Systems ist. Dabei wird jeweils nach dem Fehlermodell, dem Ort und der Dauer des injizierten Fehlers unterschieden.

---

1. VHDL: **V**ery High Speed Integrated Circuits **H**ardware **D**escription **L**anguage [Ash 90] und [IEEE 93]

In dem darauffolgenden Kapitel 5 wird dann nicht mehr das interne Signalverhalten nach einer Fehlerinjektion untersucht, sondern dessen Auswirkungen auf das jeweilige Kommunikationsprotokoll. Hierbei wird neben den Häufigkeiten der aufgetretenen Fehlertypen auch eine Aufschlüsselung der Fehlerauswirkungen gemäß den einzelnen Protokollelementen angegeben. Dabei werden auch mögliche Auswirkungen auf ein Gesamtsystem kommunizierender Prozesse und Prozessoren vorgestellt.

Das sechste Kapitel dient einer Bewertung des Verfahrens von Sieh und des Werkzeugs VERIFY. Dabei wird der Aufwand der Experimenterstellung bezüglich Speicherplatz und Zeitbedarf zur Experimentdurchführung und Experimentauswertung ermittelt. Desweiteren werden Betrachtungen zum notwendigen Detailgrad der Modellierung und der Modellierungsebene angestellt.

Im abschließenden Kapitel 7 wird dann eine Zusammenfassung und Bewertung der Ergebnisse dieser Arbeit vorgestellt und Anregungen für weiterführende Untersuchungen gegeben.

Im Anhang dieser Arbeit findet sich dann noch exemplarisch der Quelltext einer Komponente des modellierten STC104, womit der Detailgrad der Modellierung veranschaulicht werden soll. Es handelt sich dabei um die VHDL-Beschreibung einer Finite State Mashine, die zum einen als Verhaltensbeschreibung angegeben wird, zum anderen als Gattermodell, das mit Hilfe eines Synthesetools erzeugt wurde.

KAPITEL
2
Fehlermodelle und Fehlerinjektion

Mit diesem Kapitel wird eine Übersicht der bestehenden Möglichkeiten zur Fehlerinjektion gegeben. Die verschiedenen Verfahren werden dabei miteinander verglichen und bezüglich ihrer Verwendbarkeit zur Erreichung des Ziel dieser Arbeit, d.h. des Bestimmens der Auswirkungen von Hardwarefehlern auf die Kommunikationsprotokolle, bewertet. Gemäß [Sie 98] sollen die beiden folgenden Begriffe festgelegt werden:

> ***Definition: “Systemmodell”***
> *Unter dem Systemmodell versteht man die Beschreibung der Hardware, der Software und der Umgebung des eigentlichen Systems. Es beschreibt das gesamte Verhalten des modellierten Systems solange keine Fehler vorhanden sind.*
>
> ***Definition: “Fehlermodell”***
> *Das Fehlermodell beschreibt welche Fehler in einem System auftreten können und welche Auswirkungen diese haben solange sie aktiviert sind. Zu jedem Fehler wird hierbei jeweils eine Wahrscheinlichkeitsverteilung für den Auftrittszeitpunkt und für die Aktivierungsdauer angegeben. Ein “einfaches Fehlermodell” liegt vor, wenn diese Wahrscheinlichkeitsverteilungen fehlen.*

Die Begriffe “Verhaltensmodell”, “Strukturelles Modell”, Modelle auf “Register-Transfer-Ebene”, auf “Gatterebene” und auf “Layout-Ebene” werden konform zu [As h90] und [Bl e96] verwendet und beziehen sich auf die möglichen Beschreibungsebenen eines Systems in VHDL.

Nach der Vorstellung der verschiedenen Methoden zur Fehlerinjektion wird das Werkzeug VERIFY vorgestellt, mit dem die Messungen dieser Arbeit durchgeführt wurden. Den Abschluß dieses Kapitels bildet die Erläuterung der Fehlermodelle, die bei den Messungen verwendet wurden.

2.1 Methoden zur Fehlerinjektion

Um die Auswirkungen von Hardwarefehlern in digitalen Systemen zu untersuchen, wird im wesentlichen zwischen Fehlern, die während des Fertigungsprozesses der Hardware auftreten, und Fehlern, die durch Alterung oder durch äußere Umwelteinflüsse entstehen, unterschieden. Für Fehler des Fertigungsprozesses sind mächtige Testverfahren bekannt. Diese Fehler wurden für diese Arbeit jedoch nicht berücksichtigt. Das Hauptaugenmerk dieser Untersuchungen liegt also bei Fehlern, die durch Alterung bzw. durch Umwelteinflüsse, wie beispielsweise durch Alphastrahlung entstehen. Dabei ist das System schon eine geraume Zeit in erfolgreichem Einsatz und wird entweder für einen begrenzten Zeitraum (wie z.B. bei Alphastrahlung) oder beispielsweise durch Überbrückung von Leitungen permanent gestört.

Bei Hardwarefehlern handelt es sich um physikalische Ereignisse, die möglicherweise zu einem zeitlich begrenzten oder auch zu einem dauerhaften Fehlverhalten des Systems führen können. Da diese physikalischen Ereignisse in der Regel nur sehr selten und im allgemeinen nicht vorhersehbar sind, kann nicht auf Messungen an der realen Hardware in ihrer natürlichen Umgebung zurückgegriffen werden. Ein weiterer Nachteil solcher Messungen ist, daß bei den heutzutage verwendeten hochintegrierten Schaltkreisen keine praktikable Möglichkeit besteht, die tatsächliche Fehlerquelle zu ermitteln und die Ausbreitung innerhalb des Chips zu verfolgen.

Die Alternative zu der eben erwähnten Methode ist die künstliche Einbringung von Fehlern in das System, d.h. Fehlerinjektion. Dabei sind zwei Ansätze zur Fehlerinjektion zu unterscheiden: Messungen an der realen Hard- und Software, sogenannte *instrumentierte Systeme* und Messungen an einem *Systemmodell*.

2.1.1 Fehlerinjektion an der realen Hardware

Man unterscheidet Hardware-instrumentierte Systeme, bei denen physikalische Veränderungen an der Hardware durchgeführt werden, und Software-instrumentierte Systeme, die über die Zugangswege der Systemsoftware oder der Firmware die Hardware zu einem Fehlverhalten zwingen.

Ein Beispiel, bei dem über kleine Kontakte eine zusätzliche elektrische Ladung auf die Leiterbahnen aufgebracht wurden ist in [Stei 97] angegeben. Bei RIFLE [MaRS 94] und MESSALINE [ArAA 90] werden die Pins einzelner Chips auf einen bestimmten Pegel gelegt. Damit sind sowohl temporäre als auch permanente Fehler injizierbar. Über den Systemtakt und zusätzliche Hardware kann außerdem das Systemverhalten an den Pins nach erfolgter Fehlerinjektion sehr gut beobachtet werden. Hierbei steht man jedoch vor dem Problem, daß mit einer wachsenden Integration von Komponenten auf einem Chip die Fehlerinjektion an den Pins des Chip in immer geringeren Maße das interne Verhalten des Chips nach einem Fehler widerspiegelt.

Eine weitere Methode der Hardware-instrumentierten Systeme besteht in der Belastung des zu untersuchenden Systems durch äußere Störparameter, wie z.B. durch einen Schwerionenbeschuß. [GuKT 89] und [KaGT 91] führten diese Methode an mehreren Prozessoren durch und verglichen die dabei gewonnen Ergebnisse mit Fehlerinjektionsexperimenten durch elektromagnetische Strahlung, die von [KaFA 95] am MARS-System durchgeführt wurden.

Beiden Methoden ist gemein, daß sie einen erheblichen Aufwand an zusätzlicher Hardware haben und nur bei Hard- und Softwaresystemen eingesetzt werden können, die schon fertig entwickelt sind. Eine Analyse des Fehlerverhaltens in der Designphase eines Systems ist nicht möglich. Desweiteren ist es bei beiden Ansätzen nicht möglich, Aussagen über das Verhalten interner Komponenten im Fehlerfall zu machen. Verbesserungsvorschläge zur Erhöhung der Fehlertoleranz durch Maßnahmen innerhalb des Systems sind somit nicht möglich.

Bei Software-instrumentierten Systemen werden mit Hilfe von Erweiterungen der Soft- oder Firmware, Fehler in das System eingebracht. Beispiele hierfür sind FIAT [SeVS 88], das es erlaubt, den Speicherbereich eines Prozesses während seiner Laufzeit zu modifizieren und FERRARI [KaKA 92], das zusätzlich Software-Ausnahmen beim Prozessor erzeugen kann. DOCTOR [HaRo 93], Xception [CaMS 95] und EFA [EcLe 92] erlauben eine Fehlerinjektion in verteilte Systeme. Ein gravierender Nachteil dieser Fehlerinjektionssysteme ist, daß sie nur einen kleinen Teil der möglichen Fehler bzw. der möglichen Fehlverhalten der Systeme abdecken. Auch bei Software-instrumentierten Systemen ist es nicht möglich, in den frühen Designphasen eines Systems schon Ergebnisse über das Fehlerverhalten dieses Systeme in den Entwurf einfließen zu lassen.

2.1.2 Simulationsbasierte Fehlerinjektion

Bei der simulationsbasierten Fehlerinjektion wird aus der Hard- und Software des zu untersuchenden Systems ein Systemmodell generiert. Damit hat man in der Regel eine volle Beobachtbarkeit jeder Teilkomponente des Modells. Es besteht damit die Möglichkeit, sicherheitskritische Komponenten zu identifizieren und evtl. sogar schon während der Designphase bezüglich deren Fehlertoleranzeigenschaften zu verbessern.

Beispiele für Modellierungswerkzeuge mit einem integrierten Fehlermodell sind REACT [ClPr 94], ADEPT [KuKA 94] und SimPar [HeGo 95]. Mit diesen Werkzeugen lassen sich Systemmodelle aus vorgefertigten Bausteinen zusammensetzen. Für jede der Komponenten wird neben dem fehlerfreien Verhalten auch noch das Verhalten im Fehlerfall angegeben. Die bereitgestellten atomaren Komponenten stellen bei diesen Systemen jedoch meistens schon sehr komplexe Hard- und Softwaresysteme (z.B. einen Prozessor oder einen Speicher) dar. Damit ist der Detailgrad der Modellierung speziell bei hochintegrierten Systemen sehr gering.

Eine Alternative stellen diejenigen simulationsbasierten Werkzeuge dar, die direkt auf einer etablierten Hardwarebeschreibungssprache basieren. Ein Beispiel dafür ist die Injektion temporärer Fehler in einen Controller eines Jet-Antriebs auf Schaltkreisebene [ChIC 90]. Jedoch ist diese Methode bei Messungen mit mehreren tausend Fehlern aus Gründen des Rechenaufwands nicht machbar. Mit [CzSi 90] wurden in ein VERILOG-Modell eines IBM-RT-PC annähernd 19000 Fehler injiziert, um die Ausbreitung von Fehlern und ihre Auswirkungen zu untersuchen. Auch hier wurde die Anzahl der möglichen Fehler im System nur sehr klein gehalten, um den Rechenaufwand erträglich zu gestalten.

Mit MEFISTO von [JeAR 94] wurde dann zum ersten Mal die standardisierte Hardwarebeschreibungssprache VHDL verwendet, das mit Hilfe des Systemmodells ein Fehlermodell generiert. Dies basiert zum einen auf der Erweiterung des Systemmodells um sogenannte Saboteure, die Fehler in das System einbringen. Zum anderen wurde dabei die Möglichkeit von Mutanten vorgesehen, die sich im fehlerfreien Fall wie die Orginalkomponente des Systemmodells verhalten und im Fehlerfall ein davon abweichendes Verhalten aufzeigen. Desweiteren besteht bei MEFISTO die Möglichkeit, Variablen und Signale von VHDL während der Simulationszeit über Eingriffe in den Simulator zu verändern. Der Nachteil des Mutantenansatzen von MEFISTO ist die statische Natur der Mutanten. Für jedes Fehlerverhalten muß ein eigener Mutant entwickelt und in das restliche System eingebunden werden. Dies führt zu einer zeitlich nicht zumutbaren Belastung, wenn man mehrere tausend Fehler injizieren will.

Das von Sieh [Sie 98] entwickelte Werkzeug VERIFY erweitert die Idee von MEFISTO auf dynamische Mutanten mit Hilfe eines eigens dafür entwickelten Simulators für VHDL-Modelle. Der genauer Aufbau von VERIFY wird in Kapitel 2.2 vorgestellt.

2.1.3 Vergleich der Injektionsverfahren

In Tabelle 2-1 werden die vorgestellten Verfahren miteinander verglichen und bewertet. Die Bewertungskriterien (Spalten der Tabelle) sind dabei in mehrere Blöcke zusammengefaßt. Im ersten Block wird das jeweilige Werkzeug bezüglich der möglichen Modellierungsebenen überprüft. Dabei ist zu erkennen, daß außer den Methoden, die auf einer Hardwarebeschreibungssprache basieren, die jeweiligen Werkzeuge hauptsächlich auf eine Modellierungsebene beschränkt sind.

	System Level	RTL Level	Pin Level	Gatter level	Einheitliches Modell	Integriertes Fehlermodell	Detailltreue Modellierung	Verwendbar in Designphase	Hierarchische Modelle	Beobachtbarkeit	Kontrollierbarkeit	Effiziente Auswertung
ADEPT	+ +	+	-	- -	-	+	-	+ +	+ +	+ +	+ +	+/-
DEPEND	+ +	+	- -	- -	+ +	+	- -	+ +	+	+ +	+ +	-
REACT,	+ +	- -	- -	- -	- -	-	- -	+ +	- -	+ +	+ +	-
SIMPAR	+ +	+	- -	- -	+ +	+	- -	+ +	+ +	+ +	+ +	-
MEFISTO	+ +	+ +	+ +	+ +	- -	- -	+	+ +	+ +	+ +	+ +	- -
FERRARI	- -	+ +	- -	- -	- -	- -	+	- -	- -	-	+	++
FIAT	- -	+ +	- -	- -	- -	- -	+	- -	- -	-	+	+ +
DOCTOR	- -	+ +	- -	- -	- -	- -	+	- -	- -	-	+	+ +
XCEPTION	- -	+ +	- -	- -	- -	- -	+	- -	- -	-	+	+ +
EFA	- -	+ +	- -	- -	- -	- -	+	- -	- -	-	+	+ +
FINE	- -	+ +	- -	- -	- -	- -	+	- -	- -	-	+	+ +
Heavy Ion	- -	- -	- -	+ +	-	-	- -	- -	- -	- -	- -	+ +
Power Dist.	- -	- -	- -	+ +	-	-	- -	- -	- -	- -	- -	+ +
MESSALINE	- -	- -	+ +	- -	- -	- -	- -	- -	- -	-	+ +	+ +
RIFLE	- -	- -	+ +	- -	- -	- -	- -	- -	- -	-	+ +	+ +
VERIFY	+ +	+ +	+ +	+ +	+ +	+ +	+ +	+ +	+ +	+ +	+ +	+

+ +	gute Unterstützung	+	wird Unterstützt	-	kaum Unterstützung	- -	keine Unterstützung

Tab. 2-1: Vergleich der Fehlerinjetionsverfahren

Im zweiten Block sind Eigenschaften aufgeführt, die den Zusammenhang zwischen Systemmodell bzw. realer Hardware und dem Fehlermodell bewerten. So wird mit dem Begriff des einheitlichen Modells eine Aussage darüber getroffen, in wie weit das Fehler- und das Systemmodell einander entsprechen. Weitere Fragen hierbei sind die Möglichkeit der Unterstützung hierarchischer Modelle, ob das Werkzeug schon während der Designphase eingesetzt werden kann, und wie detailliert das Systemmodell die Hardware nachbildet. Im dritten Block der Eigenschaften wird die Beobachtbarkeit und Kontrollierbarkeit des Systems bzw. der Experimente bewertet. Wie im vorhergehenden Kapitel schon erwähnt, schneiden hierbei die simulationsbasierten Werkzeuge zur Fehlerinjektion am besten ab. Der Nachteil der simulationsbasierten Systeme ist zweifelsohne ihre Effizienz, da die Simulation eines detaillierten Hardwaremodells sehr viel Zeit in Anspruch nimmt.

2.2 VERIFY

Während des letzten Jahrzehnts wurde VHDL zur dominierenden Beschreibungssprache für die Hardware integrierter digitaler Schaltungen. Die Standardisierung der Sprache und ihre Beschreibungsmöglichkeiten auf den verschiedensten Modellierungsebenen hat VHDL zu einem unverzichtbaren Werkzeug für das Design digitaler Schaltungen werden lassen. Ein wichtiger Bestandteil aller Werkzeuge, die auf VHDL basieren, ist der Simulator, mit dessen Hilfe das modellierte System bezüglich der funktionellen und zeitlichen Korrektheit getestet werden kann. In der standardisierten Sprache von VHDL ist jedoch keine Beschreibung der Zuverlässigkeitsparameter von Komponenten vorgesehen.

2.2.1 Die Spracherweiterung von VHDL

Mit dem von Sieh entwickelten Werkzeug VERIFY ist es nun möglich, mit VHDL neben dem funktionellen und zeitlichen Ablauf der Ereignisse innerhalb einer Komponenten auch die zu ihr gehörigen Fehlertypen, -auswirkungen und deren Häufigkeiten zu beschreiben. Dazu war es notwendig die Sprache zu erweitern, da es sich hier um eine Beschreibung nichtdeterministischer Ereignisse (Auftrittszeitpunkte und -dauern von Fehlern) handelt. Da das ursprüngliche Konzept von VHDL durch die Beschränkung auf die Beschreibung digitaler Einheiten deterministisch ausgelegt ist, handelt es sich also um eine konzeptionelle Erweiterung der Sprache.

Eine Mehrheit der in Kapitel 2.1 vorgestellten Methoden zur Fehlerinjektion zeichnet eine Trennung von Fehlermodell und Systemmodell bzw. realer Hardware aus. Das Fehlermodell bildet bei ihnen Teile des eigentlichen Systems nach und birgt somit eine Quelle von Inkonsistenzen der Modellbeschreibungen. Neben der damit verbundenen Gefahr falscher Meßergebnisse hat man bei diesen Verfahren auch noch den zusätzlichen Aufwand, die Hardware für das Fehlermodell ein zweites Mal zu modellieren. Dieses Problem kann mit VERIFY umgangen werden, da hier der Grundsatz gilt, daß das Fehlermodell direkt in das Systemmodell integriert ist. Die Fehler werden also nicht außerhalb des Systemmodells beschrieben, sondern sind direkter Bestandteil der mit VHDL beschriebenen Komponenten. Die Beschreibung jedes zu berücksichtigenden Fehlers innerhalb einer Komponente geschieht mit Hilfe eines neuen Signaltyps. Signale dieses Typs werden im folgenden als Fehlerinjektionssignale (FIS) bezeichnet. Bei einer Plazierung der FIS in der **entity**-Beschreibung zöge jede

Änderung des Fehlermodells einer einzelnen Komponente eine Änderung des gesamten Pfades im hierarchischen Aufbau des Gesamtmodells nach sich. Diese hierarchische Abhängigkeit widerspräche jedoch dem Prinzip der Kapselung komponentenspezifischer Daten. Die Fehlerbeschreibung einer Komponente wäre über mehrere Hierarchieebenen hinaus sichtbar (bis hin zum Testbed). Aus diesem Grund wurden die FIS in den **architecture**-Teil der Komponentenbeschreibung gelegt.

Die Beschreibung des Verhaltens der Komponente bei Aktivierung eines der FIS geschieht durch die Erweiterung des Verhaltensmodells. Für jedes FIS der Komponente, wird dabei mit den üblichen Beschreibungsmöglichkeiten von VHDL das komponentenspezifische Fehlerverhalten angegeben. In Abb. 2-1 ist eine Möglichkeit angegeben, mit der ein NOT-Gatter um sein Fehlerverhalten erweitert werden kann. Es handelt sich dabei um ein *Stuck-At*-Fehlermodell, bei dem entweder der Eingang oder der Ausgang der Komponente auf dem Wert ‘0’ bzw. ‘1’ gehalten wird.

```
ENTITY not_gate IS
     PORT (    input:        IN       bit;
               output:       OUT      bit
          );
END not_gate;



ARCHITECTURE behaviour OF not_gate IS
     SIGNAL    i_sa0:    BOOLEAN INTERVAL 10000 h DURATION 5 ns;
     SIGNAL    i_sa1:    BOOLEAN INTERVAL 15000 h DURATION 5 ns;
     SIGNAL    o_sa0:    BOOLEAN INTERVAL 20000 h DURATION 5 ns;
     SIGNAL    o_sa1:    BOOLEAN INTERVAL 30000 h DURATION 5 ns;
BEGIN
          PROCESS (input, i_sa0, i_sa1, o_sa0, o_sa1) BEGIN
                    IF i_sa0 OR o_sa1 THEN
                              output <= ’1’;
                    ELSIF i_sa1 OR o_sa0 THEN
                              output <= ’0’;
                    ELSE
                              output <= NOT input AFTER 10 ns;
                    END IF;
          END PROCESS;
END behaviour;
```

Abb. 2-1: Verhaltensmodell eines NOT-Gatters mit integriertem Fehlermodell

In dem Beispiel sind diejenigen VHDL-Anweisungen fett gedruckt, die zur Beschreibung eines NOT-Gatters ohne Fehlerverhalten verwendet worden wären. Wie sich aus dem Beispiel ersehen läßt, ist die **entity**-Beschreibung, d.h. derjenige Teil, der für die Außenwelt der Komponente wichtig ist, unverändert bezüglich der Fehlerbeschreibung. Im **architecture**-Teil werden dann die FIS **i_sa0**, **i_sa1**, **o_sa0** und **o_sa1** beschrieben. Ein *Stuck-At-1* Fehler am Ausgang des NOT-Gatters (**o_sa1**) kommt gemäß der Beschreibung im Mittel alle 30000 Stunden vor und hat eine mittlere Dauer von 5 *ns*. Die Sensitivitätsliste des Prozesses, der das Verhalten des

Gatters beschreibt, wird nun um die FIS und die Verhaltensbeschreibung des Gatters gemäß der Natur des jeweiligen Stuck-At Fehlers erweitert. Das Beispiel zeigt, daß sich die FIS in der Prozeßbeschreibung wie die bisherigen VHDL-Signale verwenden lassen.

2.2.2 Aufbau und Struktur von VERIFY

Die Ermittlung des Verhaltens eines Systems im Fehlerfall geschieht mit VERIFY in drei Phasen. Eine Übersicht der Struktur von VERIFY wird in Abb. 2-2 gegeben. In der *Setup*-Phase wird ein Systemmodell der Hard- und Software zusammen mit der notwendigen Umgebung entwickelt. Dabei werden die Verhaltensbeschreibungen der Komponenten um die Fehlerbeschreibungen und die FIS erweitert. Im Schritt des Compilierens und Linkens des Quellcodes wird dann aus dieser Beschreibung ein ausführbares Simulationsprogramm, bei dem die FIS aller Komponenten des Modells für den Simulatorteil des Programms sichtbar sind.

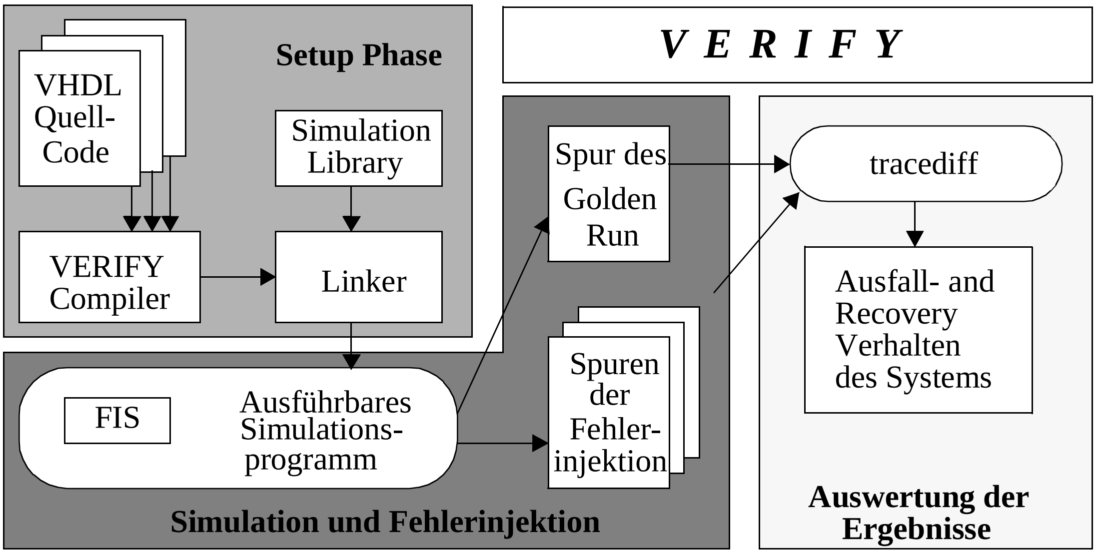


Abb. 2-2: Komponenten des Werkzeugs VERIFY

Zur Durchführung eines Experiments gibt der Benutzer von VERIFY nun an, wie viele Fehler N während einer Gesamtsimulationszeit injiziert werden sollen und wie lange das System nach jeder Fehlerinjektion beobachtet werden soll. Die Beobachtung des Systems besteht in der Aufzeichnung jeder Signaländerung im gesamten modellierten System. Neben diesen sogenannten Spuren jeder Fehlerinjektion wird eine Spur des fehlerfreien Laufs (Golden Run) während der Gesamtsimulationszeit aufgezeichnet. Neben der Gesamtsimulationszeit kann dann noch angegeben werden, ab welchem Zeitpunkt Fehler injiziert werden und wann der Zeitraum der Fehlerinjektionen beendet ist (Fehlerinjektionszeitraum T_k). Damit kann die Phase der Systeminitialisierung ohne Beeinflussung von Fehlern simuliert werden.

Ein wesentlicher Aspekt von VERIFY ist die Annahme, daß Fehler unabhängig voneinander auftreten und jeder für sich eine sehr geringe mittlere Auftrittshäufigkeit im Vergleich mit der mittleren Dauer der Fehlerauswirkungen hat. Eine Fehlerinjektion und die damit verbundene Beobachtung der Signale besteht also immer aus der Aktivierung eines einzelnen Signals aus der Liste der FIS. Gemäß dieser Einzelfehlerannahme ergibt sich bei VERIFY der in Abb. 2-3 gezeigte Experimentablauf bei N zu injizierenden Fehlern. Bei einer Gesamtsimulationsdauer von T_g besteht die Möglichkeit, die Simulation so weit voran schreiten zu lassen, bis sich das System in einem eingeschwungenen Zustand befindet (T_s). Danach werden N voneinander unabhängige Fehler bis zu einem Zeitpunkt T_e in das System eingebracht.

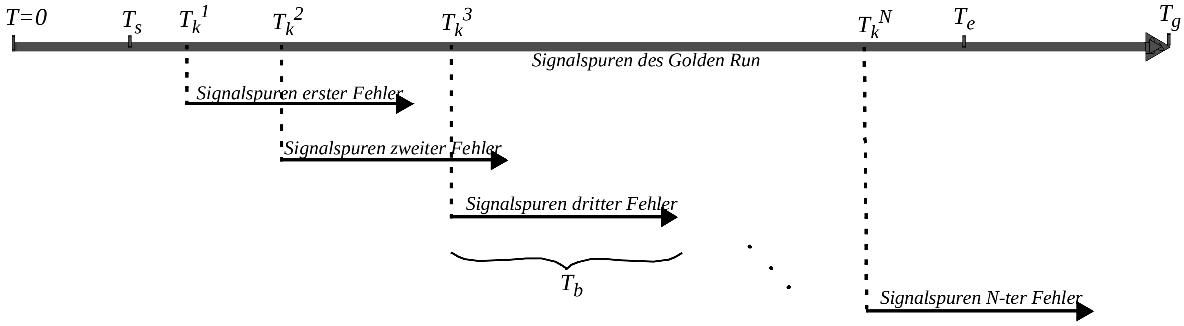


Abb. 2-3: Simulationsablauf und Signalspuren bei N zu injizierenden Fehlern

Die Aktivierungszeiten T_k^j der N Fehler ($j \in 1 \ldots N$) im Fehlerinjektionszeitraum $T_k = [T_s \ldots T_e]$ werden vom Simulator mit Hilfe eines Pseudozufallszahlengenerators gleichverteilt über T_k ermittelt. Der Typ des jeweils als nächsten zu aktivierenden Fehlers wird, ebenfalls mit Hilfe eines Pseudozufallszahlengenerators, aus der Gesamtheit der beschriebenen FIS ausgewählt. Bei dieser Auswahl wird die relative Häufigkeit jedes Fehlers der FIS berücksichtigt. Hat ein Fehler F_i der FIS eine doppelt so große mittlere Auftrittshäufigkeit wie ein zweiter Fehler F_j, so ist die Wahrscheinlichkeit für eine Aktivierung des Fehlers F_i doppelt so groß wie die von F_j. Bei der Aktivierung eines Fehlers F_j wird zum Zeitpunkt T_k^j eine vollständige Kopie des Simulationprozesses des Golden Run erzeugt. Gemäß der Beschreibung des Fehlers F_j wird darauf hin die Fehleraktivierungsdauer bestimmt, und das Fehlerinjektionssignal in der Kopie des fehlerfreien Simulationsprozesses während dieses Zeitraums aktiviert. Im Anschluß daran werden die Auswirkungen des Fehlers, d.h. die Spuren aller im Simulationssystem befindlicher Signale für eine vorgegebene Beobachtungsdauer T_b aufgezeichnet. Diese Spuren werden ausschließlich in der Kopie des Simulationsprozesses erzeugt. Der Simulationsprozeß des Golden Run ist für diese Zeit gestoppt. Nach Beendigung der Beobachtungsdauer T_b wird der Simulationsprozeß des Golden Run wieder aktiviert und fährt mit der Berechnung des fehlerfreien Systemverhaltens zum Zeitpunkt T_k^j fort. Ein Fehler und die durch ihn bedingten Auswirkungen macht sich also ausschließlich in der Kopie des Simulationsprozesses des Golden Run bemerkbar und ist weder im Golden Run, noch in den Signalspuren anderer Fehler sichtbar.

Das in Abb. 2-3 beschriebene Verfahren stellt die *Multi-Threaded-Fault-Injection* dar, das die Gesamtsimulationszeit im Vergleich zu herkömmlichen Fehlerinjektionsverfahren gering hält [Sie 98]. Mit diesem Beschleunigungsverfahren ist es also möglich, das Fehlerverhalten realer Systeme mit oft sehr großen VHDL-Modellen zu untersuchen.

In der dritten Phase wird nun von VERIFY jede Spur einer Fehlerinjektion mit der Spur des Golden Run verglichen. Sind alle Signale des Systemmodells zu einem beobachteten Zeitpunkt nach der Fehlerinjektion gleich den Signalen des fehlerfreien Laufs, so hat sich das System selbständig von dem Fehler erholt. Dabei wird von dem Auswertewerkzeug von VERIFY auch berücksichtigt, daß es zu einer zeitlichen Verzögerung zwischen den Signalabfolgen des fehlerfreien und des fehlerbehafteten Simulationslaufs kommen kann. Wenn bis zum Ende des Beobachtungszeitraums noch Signalunterschiede zu finden sind, gilt das System als dauerhaft geschädigt (Crash).

2.2.3 Bestimmung der Anzahl der zu injizierenden Fehler

Die Angaben der Auftrittshäufigkeiten und -dauern für jeden Fehler der FIS sind statistischer Natur und VERIFY injiziert gemäß diesen Angaben den jeweils nächsten Fehler. Damit ergibt sich als wesentliches Problem der Vorbereitung eines Experiments, wie viele Fehler in das System eingebracht werden sollen. Es gilt also, eine Formel anzugeben, mit der die Konfidenz der Ergebnisse ermittelt werden kann.

Sei X eine Zufallsvariable, EX ihr Erwartungswert und DX die Varianz der Zufallsvariablen. Dann gilt gemäß der Tschebyscheffschen Ungleichung für jedes $\varepsilon > 0$:

$$P(|X - EX| \geq \varepsilon) \leq \frac{DX}{\varepsilon^2}$$

Bei einem Fehlerinjektionsexperiment mit einem Umfang von n voneinander unabhängigen Fehlern wobei es in m Fällen zu einem Crash kommt, kann man die Binomialverteilung mit den Parametern n und p annehmen, wobei p die reale Auftrittswahrscheinlichkeit des Ereignisses Crash angibt. Für diesen Fall gilt $EX = np$ und $DX = np(p - 1)$ und somit erhält man für die relative Häufigkeit *m/n* des Ereignisses (vergl. auch [Si e98]):

$$P\left(\left|\frac{m}{n} - p\right| \geq \varepsilon\right) \leq \frac{p(1-p)}{\varepsilon^2 n} \leq \frac{1}{4\varepsilon^2 n}$$

Dies bedeutet, daß die Wahrscheinlichkeit, daß eine gemessene relative Häufigkeit *m/n* eines Ereignisses von der realen Auftrittswahrscheinlichkeit p um mehr als den Betrag ε abweicht, kleiner ist als $1/(4\varepsilon^2 n)$ (bei n Tests). In der folgenden Abb. 2-4 ist für verschiedene Expermientgrößen (d.h. Anzahl der Fehlerinjektionen) die Wahrscheinlichkeit angegeben, mit der eine gemessene relative Häufigkeit (z.B. von Crashfehlern) von der realen Auftrittswahrscheinlichkeit um weniger als ein vorgegebener Wert ε abweicht.

Aus der Abb. 2-4 ist zu erkennen, daß man mit *5000* voneinander unabhängigen Fehlerinjektionen schon mit fast *95%* Wahrscheinlichkeit davon ausgehen kann, daß ein gemessener Wert einer relativen Häufigkeit von der realen Auftrittswahrscheinlichkeit um weniger als 0,03 abweicht.

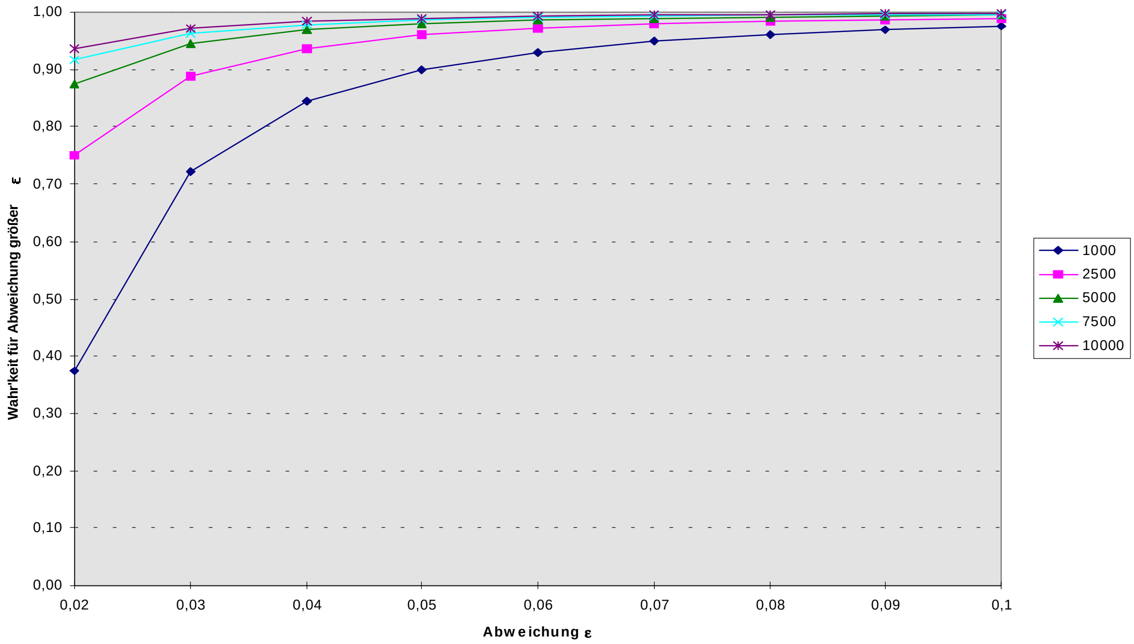


Abb. 2-4: Wahrscheinlichkeit für Abweichung der gemessenen relativen Häufigkeit von der realen Auftrittswahrscheinlichkeit bei verschiedenen Experimentgrößen

2.3 Fehlermodelle

Wie in Kapitel 1.1 schon erwähnt wurde, sind die beiden Router (ATM und STC104) für diese Arbeit auf Gatterebene modelliert worden. Alle modellierten Komponenten der Router sind dabei jeweils auf einem einzigen Chip integriert. Fehler externer Busse oder Platinenfehler sind also nicht zu berücksichtigen.

2.3.1 Stuck-At Fehler der Gatter

Ein auf Gatterebene anerkanntes Fehlermodell sind die sogenannten Stuck-At-Fehler. Mit den Stuck-At-1 Fehlern wird dabei der Wert an einem der Ein- oder Ausgänge des zu beschreibenden Gatters für die Dauer des Fehlers auf den Wert ‘1’ gelegt. Analoges Verhalten gilt hierbei für Stuck-At-0 Fehler. Jede Gatterbeschreibung wird hierbei so erweitert, daß an allen Eingangs- und Ausgangsleitungen jeweils ein Stuck-At-0 Fehler oder ein Stuck-At-1 Fehler auftreten kann. Eine Liste der verwendeten Gatter ist bei der Beschreibung des Systemmodells in Kapitel 3 mit Tabelle 3-1 angegeben.

Die Experimente sind hierbei so instrumentiert, daß neben temporären Fehlern auch permanente Stuck-At Fehler in das System eingebracht werden können. Bei den permanenten Stuck-

At Fehlern wird dazu die Dauer des zu aktivierenden Fehlers bei einer Injektion außer acht gelassen und der Fehler während des gesamten Beobachtungszeitraums aktiv gehalten. Damit ist es nicht möglich, die Zeit bis zu einem Recovery des Systems zu untersuchen, da ein vollständig fehlerfreier Zustand[1] nie erreicht werden kann. Bei Experimenten mit permanenten Fehlern wurde aus diesem Grund auch nur deren Auswirkungen auf das Verhalten der Kommunikationsprotokolle untersucht. Die Auswirkungen permanenter und temporärer Fehler auf das jeweilige Protokoll ist nicht Bestandteil von VERIFY und des Fehlermodells und wird in Kapitel 5 genauer beschrieben.

Die Notwendigkeit der Untersuchung des Systems im Falle temporärer Fehler läßt sich außer durch die Annahme einer Einwirkung energiereicher Strahlung (z.B durch Alpha-Strahlung) auch durch Untersuchungen von elektrostatischer Entladung belegen [Mard 86][BhMc 83]]. Hierbei läßt sich die Auswirkung elektrostatischer Entladungen in zwei Kategorien unterteilen. Zum einen handelt es sich um Fehler, die nur den Kontrollfluß der Anwendung beeinträchtigen[2], die Hardware der Systemkomponenten jedoch nicht schädigen. Dieser Fehlertyp tritt meist unmittelbar nach der elektrostatischen Entladung ein, kann jedoch auch verzögerte Auswirkungen auf das Systemverhalten aufweisen, falls ein größerer Zeitraum zwischen der Entladung und der Verwendung der betroffenen Komponente liegt.

Im zweiten Fall handelt es sich um hardwareschädigende Auswirkungen. Untersuchungen haben gezeigt, daß die Energie der Entladung einen Kanal zwischen zwei leitenden Schichten eines integrierten Schaltkreises erzeugen kann [PiDu 81]. Haben die Leiter eine unterschiedliche Polarität (z.B. bei P-N Übergängen), so kann leitendes Material in den Kanal wachsen. In dem Moment, wenn der Kanal leitend wird, schmilzt das Material im Kanal wieder auf, und unterbricht den Kurzschluß (Abb. 2-5). Dieses Verhalten kann sich mehrfach wiederholen. Im Gegensatz zu Fehlern ohne hardwareschädigende Auswirkungen kann zwischen der elektrostatischen Entladung und dem Auftreten des ersten Kurzschlusses ein längerer Zeitraum liegen, so daß die Komponente schon längere Zeit im Einsatz gewesen sein kann. Wir haben es hierbei also mit temporären Fehlern, im Falle mehrfacher Schmelzvorgänge sogar mit intermittierenden[3] Fehlern zu tun.

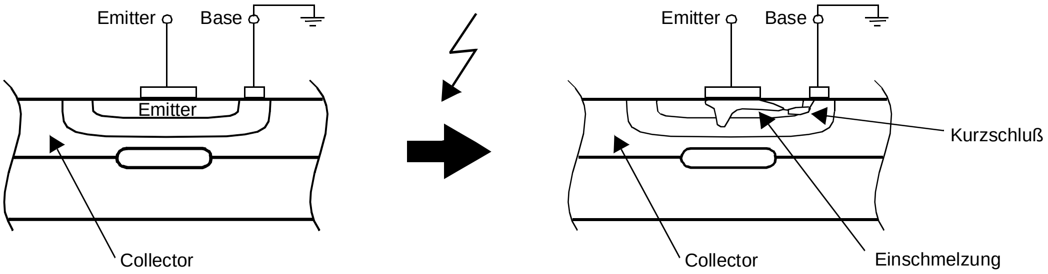


Abb. 2-5: Schmelzvorgang am P/N Übergang eines Transistors durch Elektrostatische Entladung

---

[1] gekennzeichnet durch identische Werte aller Signale beim Fehlerinjektionslauf und beim Golden Run über die Gesamtrestlaufzeit des Systems.

[2] zum Beispiel durch Leitungsstörungen auf Grund eines durch die Entladung entstandenen magnetischen Feldes

[3] intermittierende Fehler sind Fehler, die in unregelmäßigen Abständen wiederkehren (z.B. Wackelkontakt)

2.3.2 Übersprechfehler auf Gatterebene

Neben dem bekannten Stuck-At-Fehlermodell wird in dieser Arbeit ein weiteres Fehlermodell vorgestellt, das ebenso auf physikalischen Ereignissen innerhalb des Chips basiert. Hierbei wird davon ausgegangen, daß sich die hierarchische Struktur, die beim Entwurf eines Gattermodells entsteht, in räumlich zusammenhängenden Blöcken beim Layout des Chips widerspiegelt. Aus dieser Annahme folgt, daß Signalleitungen und damit physikalische Leitungen zwischen diesen Blöcken zumindest teilweise nebeneinander liegen. Daraus läßt sich nun ein weiteres Fehlermodell entwickeln, bei dem es zu einem Übersprechen der Signalwerte von nebeneinander liegenden Leitungen kommen kann.

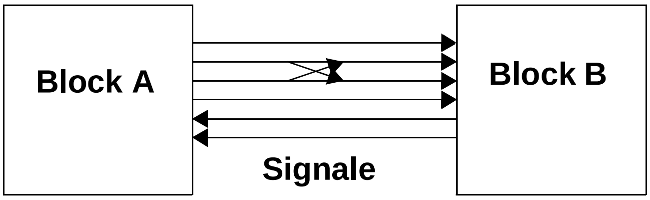


Abb. 2-6: Beispiel für Übersprechfehler

In dem in Abb. 2-6 angegebenen Beispiel kommt es bei zwei Leitungen, die Signale von Block A zu Block B transportieren zu einem Übersprechen. Das Ergebnis dieses Übersprechens ist abhängig von den Werten, die an den Leitungen anliegen und von der Stärke des jeweiligen Signaltreibers. Liegen bei VHDL zwei unterschiedliche Signalwerte an derselben Leitung an, so muß für den resultierenden Signalwert eine **resolution**-Funktion angegeben werden. Damit verhält sich der resultierende Wert beider Signale ähnlich zu dem in VHDL bekannten Signaltyp des **Busses** bei dem auch mehrere Treiber einen einzelnen Signalwert bestimmen können. Für den Fall eines dominierenden Signaltreibers (des Wertes ‘1’) ist in Tabelle 2-2 und für den Fall einer dominierenden Masse ist in Tabelle 2-3 die **resolution**-Funktion angegeben. Grundlage für diese Funktionen ist die von IEEE standardisierte 9-wertige Logik **std_ulogic**, die heutzutage weit verbreitet im Hardwaredesign eingesetzt wird [Ble 96]. Die einzelnen Werte haben dabei folgende Bedeutung:

• U	Unbekannter Wert; wird zumeist in der Initialisierungsphase der Signale verwendet
• X	Unbestimmt; Bsp: auf einem Bus wird gleichzeitig eine ‘1’ und eine ‘0’ getrieben
• 0	Eine starke ‘0’, d.h. Verbindung mit der Masse
• 1	Eine starke ‘1’, d.h. starker Signaltreiber
• Z	Hohe Impedanz; dieser Wert beeinflußt keine anderen Werte
• W	Schwach unbestimmt
• L	Eine schwache ‘0’
• H	Eine schwache ‘1’
• -	‘Don’t care’

Für die System- und Fehlermodellierung werden hierbei jedoch nur die Werte ’U’, ’0’, ’1’, ’Z’, ’L’ und ’H’ betrachtet. Der ’dont care’-Wert ’-’ spielt für unsere Betrachtungen keine Rolle. Die

Werte 'X' und 'W' entstehen im (eingeschwungenen) System nur im Falle einer fehlerhaften Auflösung an einem Bussignal. Diese beiden Werte würden sich im System sofort verbreiten und sich mit hoher Wahrscheinlichkeit in einem Registerwert des Systems niederschlagen. Damit hätte man es in diesem Fall fast immer mit permanenten Fehlern zu tun, was jedoch nicht dem realen Verhalten des Systems entspräche. Bei Übersprechfehlern werden diese beiden 'unbestimmt'-Werte in jedem Fall von einem Folgegatter als '0', '1', 'L' oder 'H' interpretiert. Zusätzlich können wir davon ausgehen, daß sich im eingeschwungenen System keine 'Unbekannt'-Werte aus der Initialisierungsphase befinden. Aus diesem Grund ist in den beiden Auflösungsfunktionen von Tabelle 2-2 und Tabelle 2-3 der für unsere Betrachtungen relevante Bereich grau unterlegt. Die restlichen Werte der Funktionstabelle sind der Vollständigkeit halber angegeben, spielen aber bei der Simulation des Systems im Fehlerfall keine Rolle.

	U	**X**	**0**	**1**	**Z**	**W**	**L**	**H**	**-**
U	U	U	U	1	U	U	U	U	U
X	U	X	X	1	X	X	X	X	X
0	U	X	0	1	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	H	X
L	U	X	0	1	L	W	L	H	X
H	U	X	0	1	H	H	H	H	X
-	U	X	0	1	X	X	X	X	X

Tab. 2-2: Auflösung bei einem dominierenden Signaltreiber einer '1'

	U	**X**	**0**	**1**	**Z**	**W**	**L**	**H**	**-**
U	U	U	0	U	U	U	U	U	U
X	U	X	0	X	X	X	X	X	X
0	0	0	0	0	0	0	0	0	0
1	U	X	0	1	1	1	1	1	1
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	L	W	X
L	U	X	0	1	L	L	L	L	X
H	U	X	0	1	H	W	L	H	X
-	U	X	0	1	X	X	X	X	X

Tab. 2-3: Auflösung bei einer dominierenden Masse

Zur Modellierung dieser Fehler mit VERIFY wurde bei allen Komponenten mit Subkomponenten zwischen den intern erzeugten Signalen und den **port**-Signalen eine zusätzliche Komponente **bridging_wires** gelegt. In dieser Komponente kommt das Fehlermodell der Übersprechfehler zum tragen. Die Werte nebeneinander liegender Leitungen werden bei aktiviertem FIS gemäß den obigen Tabellen verändert.

2.3.3 Fehler auf algorithmischer Ebene

Als letztes Fehlermodell wird in dieser Arbeit eine Fehlerbeschreibung für ein rein auf algorithmischer Ebene beschriebenes ATM-Knockout Modell verwendet. Die Idee hierbei ist eine Untersuchung der Unterschiede der Fehlerauswirkungen zwischen dem sehr nah an der Hardware angelehnten Stuck-At-Fehlermodell und der Beschreibung der Fehler auf algorithmischer Ebene. Grundlage dieses Fehlermodells ist dabei die Arbeit von Bogendörfer [Bog 96], der ein System- und Fehlermodell des Knockout-Routers auf algorithmischer Ebene entwickelt hat. Hierbei wird von der Annahme ausgegangen, daß die Ein- und Ausgänge algorithmisch beschriebener Komponenten in ihrer physikalischen Realisierung die gleichen Signalwerte haben wie in der algorithmischen Beschreibung. Damit läßt sich nun das Stuck-At Fehlermodell auf die Ein- und Ausgangssignale aller algorithmisch beschriebener Komponenten anwenden. Für eine ausführlichere Diskussion zu diesem Fehlermodell und vorhandenen Alternativen sei auf Kapitel 6.2.1 dieser Arbeit verwiesen.

KAPITEL
3
Systemmodellierung

In diesem Kapitel werden die zur Fehlerinjektion verwendeten Systeme vorgestellt. Dabei wird neben der Vorstellung der digitalen Vermittlungseinheiten (Switches) und der durch sie verwendeten Protokolle besonders auf die Modellierung der Systeme eingegangen. Den Abschluß dieses Kapitels bildet eine Vorstellung der Lastmodelle, welche die beiden Switches mit einem protokoll- bzw. einem anwendungstypischen Datenstrom versorgen.

Für die Untersuchung des Fehlerverhaltens wurden für diese Arbeit zwei Switches ausgesucht, die sich wesentlich in ihrem Protokollverhalten unterscheiden. Es handelt sich hierbei einmal um den INMOS STC104, der als asynchroner Paketswitch einen ‘Wormhole-Routing’-Algorithmus implementiert und dessen Datenaustausch mit Kommunikationspartnern durch einen Flußkontrollmechanismus geregelt wird. Das Protokoll des STC104 basiert auf dem gleichzeitig entwickelten Protokoll für die Kommunikation zum INMOS T9000 Transputer.

Im Gegensatz dazu dient der zweite Switch zur Weiterleitung von ATM-Zellen. Obwohl das ATM-Protokoll auch einen asynchronen Datenaustausch vorschreibt, existiert hier gemäß der Protokollspezifikation explizit keine Flußkontrolle auf den unteren Kommunikationsschichten. Beim ATM-Protokoll ist es sogar erlaubt, einzelne ATM-Zellen ohne eine weitere Fehlermeldung aus dem Datenstrom zu entfernen, wenn es zu einem Kommunikationsengpaß an einem Switch kommt. Ein Flußkontrollmechanismus existiert nur auf den für ATM spezifizierten End-zu-End Protokollen AAL (**A**TM **A**daption **L**ayer) welche in Kapitel 3.3.2 genauer vorgestellt werden.

3.1 Modellierung des INMOS STC104

Der von INMOS entwickelte Switch STC104 ist ein asynchroner Paketswitch mit 32 bidirektionalen Links, die jeweils bis zu 100 Mbits/s seriell übertragen können. Im folgenden wird neben den verschiedenen Protokollebenen die Struktur des STC104 und die bei der Modellierung berücksichtigten Aspekte vorgestellt.

3.1.1 Die Protokollebenen des STC104

Das von der Hardware des T9000 im Zusammenhang mit dem STC104 vorgegebene Kommunikationsprotokoll kann in drei Protokollebenen eingeteilt werden: *Paketebene*, *Tokenebene* und *Bitebene* die im folgenden beschrieben werden.

Die Paketebene

Eine Nachricht, die zwischen zwei Prozessoren über ein Netzwerk aus STC104 Switches geleitet werden soll, wird zunächst in Datenpakete der maximalen Größe von 32 Byte Nutzdaten aufgeteilt. Jedem dieser Pakete wird dabei ein Header vorangestellt, der den Empfänger kennzeichnet. Das Kommunikationsschema basiert auf virtuellen Kanälen zwischen je zwei kommunizierenden Prozessen. Der Header jedes Pakets kennzeichnet hierbei eindeutig einen Empfangsprozeß im Gesamtsystem, wodurch es auf diesen Protokollebenen zu keinem Multicast kommen kann. Eine Nachricht von einem Sender an mehrere Empfänger weiterzuleiten muß auf höheren Protokollebenen realisiert werden, wobei zwischen dem Sendeprozeß und jedem der Empfängerprozesse jeweils ein eigener Kanal aufgebaut werden muß. Damit ist bei diesem Protokoll ausgeschlossen, daß ein Vermittlungsknoten eingehende Datenpakete vervielfältigt, um eine Nachricht an mehrere Empfänger weiterzuleiten.

Mit dem Kennzeichner *End-Of-Packet* wird ein Paket abgeschlossen, dem ein weiteres Paket derselben Nachricht folgt. Der Kennzeichner *End-Of-Message* wird an das letzte Paket der zu versendenden Nachricht angehängt. Sobald der Empfangsprozessor ein Paket empfangen hat, quittiert er dieses mit einem Acknowledge-Paket, das aus einem Header besteht, der den Sender der Nachricht kennzeichnet, und einem direkt darauf folgenden *End-Of-Packet* Kennzeichner. Da in einem Acknowledge-Paket keine Datenbytes vorhanden sind, ist die Eindeutigkeit eines Acknowledge-Paktes gewährleistet. In Abb. 3-1 wird das Protokoll der Paketebene noch einmal graphisch veranschaulicht.

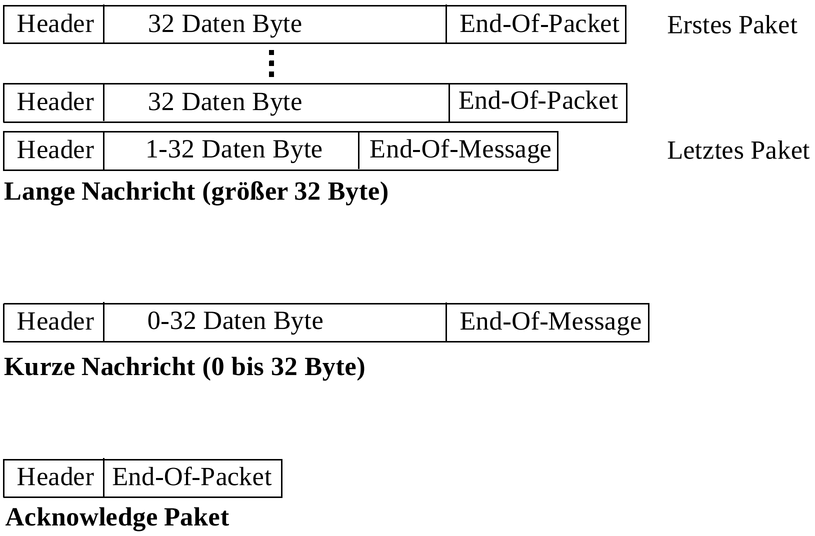


Abb. 3-1: Protokoll der Paketebene

Die Tokenebene

Der Header und die Nutzdaten eines zu versendenden Pakets werden in der darunter liegenden Tokenebene in sogenannte Datentoken mit jeweils genau acht Bit aufgeteilt. Die Endekennungen der Pakete werden auf der Tokenebene durch die Kontrolltoken *End-Of-Packet Token* (EOP) und *End-Of-Message Token* (EOM) umgesetzt. Neben diesen nachrichtenbasierten Token existieren beim Protokoll der Tokenebene noch drei weitere Tokentypen, die zur Flußkontrolle zwischen zwei direkt benachbarten Hardwareeinheiten benötigt werden (siehe Abb. 3-2). Nachdem der initiale Verbindungsaufbau zwischen zwei STC104 bzw. zwischen einem STC104 und einem T9000 erfolgt ist, wird zur Erkennung von Verbindungsunterbrechungen ein kontinuierlicher Datenstrom zwischen den Hardwareeinheiten aufrecht erhalten. Ein *Null Token* (NUL) wird dabei immer dann gesendet, wenn kein anderes Token (Daten, EOP, EOM, FCT) versendet werden kann. Ein *Flow-Control Token* (FCT) dient hierbei der Vermeidung von Überläufen von internen Puffern der Kommunikationspartner (STC104 bzw. T9000). Jedem der 32 eingehenden Links des STC104 ist ein Puffer zugeordnet der bis zu 20 Token aufnehmen kann. Wenn mindestens acht freie Plätze in dem Puffer vorhanden sind, wird der Hardwareeinheit, die mit diesem Link verbunden ist, ein FCT-Token gesendet. Der Sendeeinheit ist damit erlaubt, acht weitere Token über diesen Link zu senden. Sobald der Puffer voll belegt ist (weil z.B. der Ausgangslink noch nicht frei ist), muß die Sendeeinheit bis zum Erhalt des nächsten FTC-Token NUL-Token senden.

Tokentyp	Abkürzung	Kodierung
Daten Token	**-**	P 0 D D D D D D D D
Flow-Control Token	**FCT**	P 1 0 0
End-Of-Packet Token	**EOP**	P 1 0 1
End-Of-Message Token	**EOM**	P 1 1 0
Escape Token	**ESC**	P 1 1 1
Null Token	**NUL**	**ESC** P 1 0 0

P = Paritybit D = Datenbit

Abb. 3-2: Protokoll der Tokenebene

Die Bitebene

Jedes Token besteht aus einem Paritybit, das eine Fehlererkennung über die Bits des vorhergehenden Tokens ermöglicht, einem Bit das den Tokentyp angibt (Kontrolltoken oder Datentoken), und je nach Tokentyp weitere Bits mit Nutzinhalt. In Abb. 3-3 sind beispielhaft zwei aufeinander folgende Token dargestellt, wie sie auf der Bitebene versendet werden. Das Protokoll der Bitebene verwendet dabei zwei Leitungen pro Richtung (Data und Strobe), um Bits zu übertragen. Da jeder Link bidirektional ausgerichtet ist, stehen also pro Link 4 Leitungen, d.h. je zwei Data- und Strobe-Paare, die auch DS-Links genannt werden, zur Verfügung. Durch das Strobe-Signal wird neben den Bits der verschiedenen Token auch noch ein Clock-Signal kodiert.

Das DS-Link-Protokoll sichert zu, daß jeweils nur eine der beiden Leitungen eine Flanke zu jedem übertragenen Bit aufweist. Der Wert des auf der Datenleitung liegenden Signals gibt dabei den Wert des zu übertragenden Bits an. Das Strobe-Signal ändert seinen Wert immer dann, wenn das Datensignal gleich bleibt. Der damit kodierte Takt ermöglicht der empfangenden Hardwareeinheit eine asynchrone Erkennung der übertragenen Daten. Da die verschiedenen Token unterschiedlich viele Bits beinhalten, überdeckt ein Paritybit immer die Nutzbits des vorhergehenden Token und das Parity- und das Tokentypbit des aktuellen Token (vergl. Abb. 3-3). Damit wird ermöglicht, daß auch ein einzelner Bitfehler im Tokentypbit erkannt werden kann.

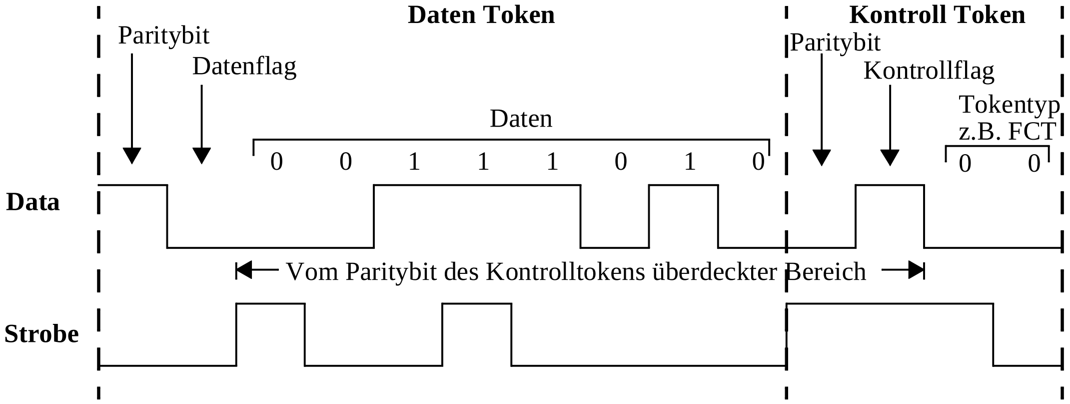


Abb. 3-3: Protokoll der Bitebene

3.1.2 Der Strukturelle Aufbau des STC104

3.1.2.1 Übersicht

Der interne Aufbau des STC104 kann in die folgenden strukturellen Einheiten unterteilt werden: *Command Processor*, *Crossbar* und 32 *Links* (siehe Abb. 3-4: auf Seite 25). Über zwei zusätzliche Links, die direkt mit dem *Command Processor* des STC104 verbunden sind, kann der Switch konfiguriert werden. Diese Funktionalität wird jedoch nur während der Initialisierung des Switches benötigt. Daneben hat der *Command Processor* noch eine überwachende Funktion, d.h. falls einer der 32 Kommunikationslinks während des normalen Betriebs einen Fehler erkennt, wird dies dem Kontrollprozessor mitgeteilt und der betroffene Link wird automatisch abgeschaltet. Jeder Link kann mit bis zu 100 Mbits/s vollduplex Daten übertragen, wobei ausschließlich das in Kapitel 3.1.1 vorgestellte Protokoll verwendet wird.

Um die Latenzzeit der Kommunikation so gering wie möglich zu halten, wird zur Weiterleitung der Daten ein *Wormhole-Routing* Algorithmus eingesetzt. Anstatt jeweils ein gesamtes Paket zwischenzuspeichern, wird die Routingentscheidung in dem Moment getroffen, wenn der Header des Pakets eingetroffen ist. Der Rest des Paketes muß also zur Auswahl des Ausgangslinks noch gar nicht im Switch eingetroffen sein. Damit ist es möglich, daß ein Paket sich gleichzeitig in mehreren Switches befindet. Sobald das letzte Token eines Pakets den Switch verlassen hat, schaltet dieser den Ausgangslink wieder frei für ein neues, evtl. von einem

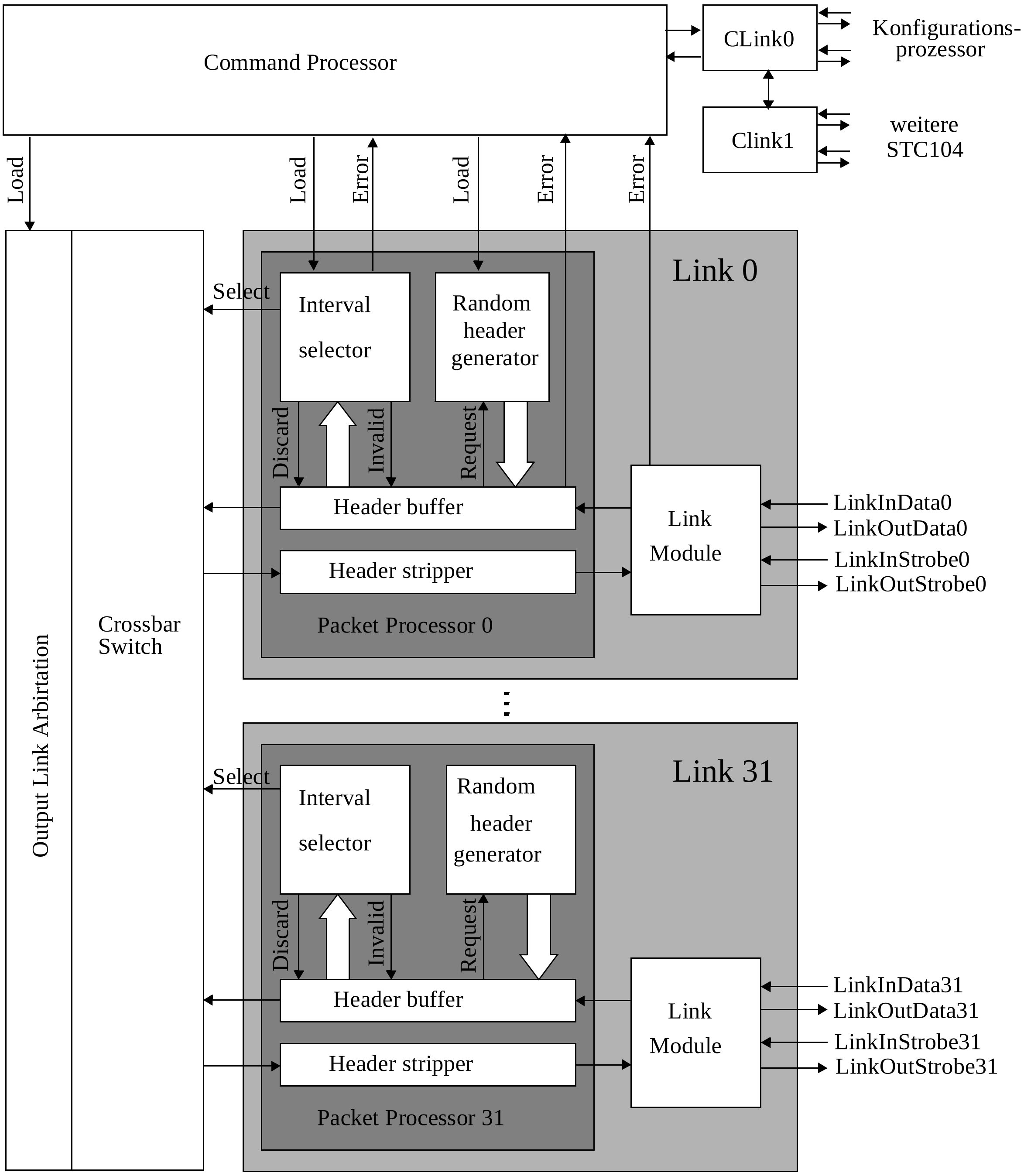


Abb. 3-4: Struktureller Aufbau des STC104 (Gesamtüberblick)

anderen Eingangslink kommendes Paket. Die Routingentscheidung wird über einen sogenannten *Interval-Labeling* Algorithmus gefällt. Dazu wird jeder Empfangsprozeß im Netzwerk mit einer eindeutigen Nummer identifiziert. Jedem Ausgangslink eines Switches wird dabei ein (numerische) Intervall zugeordnet, das angibt welche Empfangsprozesse von von diesem Ausgangslink erreichbar sind.

3.1.2.2 Der Linkaufbau

Jeder der 32 Kommunikationslinks des STC104 ist identisch aufgebaut. Die Zuordnung der Ausgangslinks geschieht während der Initialisierungsphase über die Konfigurationsregister des Switches. Eine detaillierte Übersicht der Konfigurationsregister kann in [SGS 95] gefunden werden. Die über ein DS-Link eingehenden Daten werden zuerst vom *Link-Modul* des Links übernommen (siehe Abb. 3-5). Dort wird zuerst die Parityüberprüfung im *Data-Valid-Analyzer* durchgeführt und im fehlerfreien Fall die Bits an das *Shiftregister* weitergeleitet, wo sie zu Token zusammengefaßt werden. Diese Token werden dann vom *Controller* des *Link-Moduls* ausgewertet. Handelt es sich um ein FCT-Token, wird dies unverzüglich dem *Data-Out-Manager* mitgeteilt, der damit weitere 8 Token an die mit diesem Link verbundene Einheit schicken kann. Nach dem Ausfiltern der NUL-Token werden die Pakettoken (DATA, EOP, EOM) einem *FIFO*-Puffer übergeben, der maximal 20 Token speichern kann. Der *Controller* ist über 2 Register mit dem *Command Processor* verbunden, über die er konfiguriert wird (Start, Reset, etc.) bzw. mit deren Hilfe er Nachrichten über den Linkstatus geben kann (Fehler, Verbindung unterbrochen, etc.). Sobald der *FIFO*-Puffer des *Link-Moduls* weitere 8 Token aufnehmen kann, wird über den *Controller* eine Nachricht an den *Data-Out-Manager* geleitet, der dann priorisiert ein FCT-Token in den Ausgangsdatenstrom einfügt. Der für das Senden der Daten verantwortliche *Data-Out-Manager* fügt neben den FCT-Token auch noch NUL-Token in den Datenstrom ein, falls die mit dem Link verbundene Einheit keinen Platz mehr in ihrem *FIFO*-Puffer hat (es wurde kein FCT-Token empfangen) oder falls keine Daten für den Ausgangslink vorhanden sind.

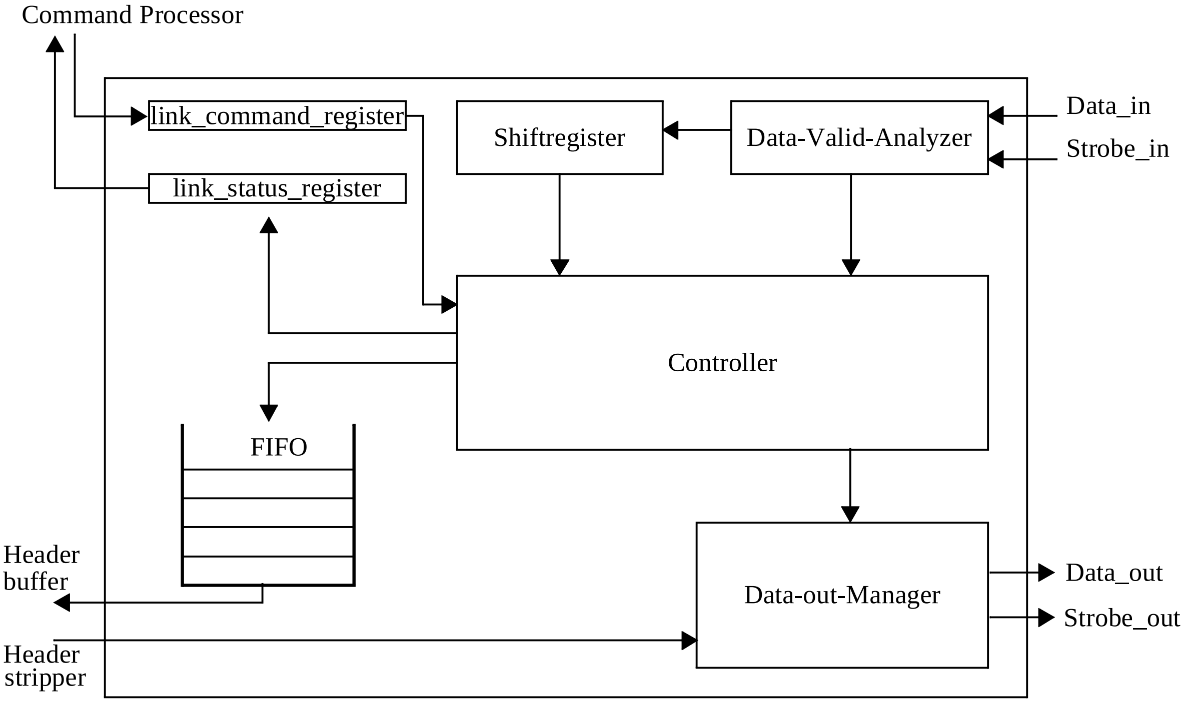


Abb. 3-5: Struktur des Link-Moduls

Vom *FIFO*-Puffer des *Link-Moduls* werden die Token dem *Header-Buffer* übergeben, der auswählt, welches der Datentoken ein Header ist und diese Token an den *Interval-Selector* übergibt. Dort wird dann eine Verbindung zum passenden Ausgangslink hergestellt, bzw. die Weiterleitung der Token dieses Eingangslinks solange blockiert, bis der Ausgangslink zugeteilt wird. Desweiteren teilt er dem *Crossbar* des STC104 mit, wenn das letzte Token (EOP oder EOM) des aktuellen Paktes versendet worden ist. Dieser kann daraufhin die zuvor aufgebaute Verbindung zwischen Eingangslink und Ausgangslink wieder freigeben. Zur Vermeidung einer Überlast auf bestimmten Verbindungsstrecken kann der *Header-Buffer* so konfiguriert werden, daß dem aktuellen Paket über den *Random-Header-Generator* ein zufällig bestimmter Empfänger innerhalb eines vorgegebenen Intervalls vorangestellt wird bevor das Paket dem *Interval-Selector* übergeben wird.

In Abb. 3-6 wird der strukturelle Aufbau des *Interval-Selectors* aufgezeigt. Für jeden eintreffenden Header wird mit Hilfe von *Comparatoren* geprüft, ob er in einem der 36 möglichen Intervalle liegt. Diese Intervalle werden bei der Konfiguration des Links festgelegt. Für jedes der Intervalle kann angegeben werden, welcher Ausgangslink für Pakete mit dieser Empfängeradresse bestimmt ist (OutLink-ID) und ob das Intervall gültig ist. Ist das gewählte Intervall ungültig gesetzt, wird das Paket verworfen und es kommt zu einer Fehlermeldung an den *Command Processor*. Weiterhin kann mit einem Flag angegeben werden, daß dieser Header von einem anderen Switch mit einem Zufallsheader versehen worden ist (s.o.), worauf der *Header-Buffer* diesen Zufallsheader entfernt und dem *Interval-Selector* die nächsten Token des Paktes als Header anbietet. Nachdem der endgültige Ausgangslink bestimmt worden ist, fordert der *Link-Selector* beim Arbitrierungsmechanismus des *Crossbars* eine Verbindung zwischen Eingangslink und Ausgangslink an.

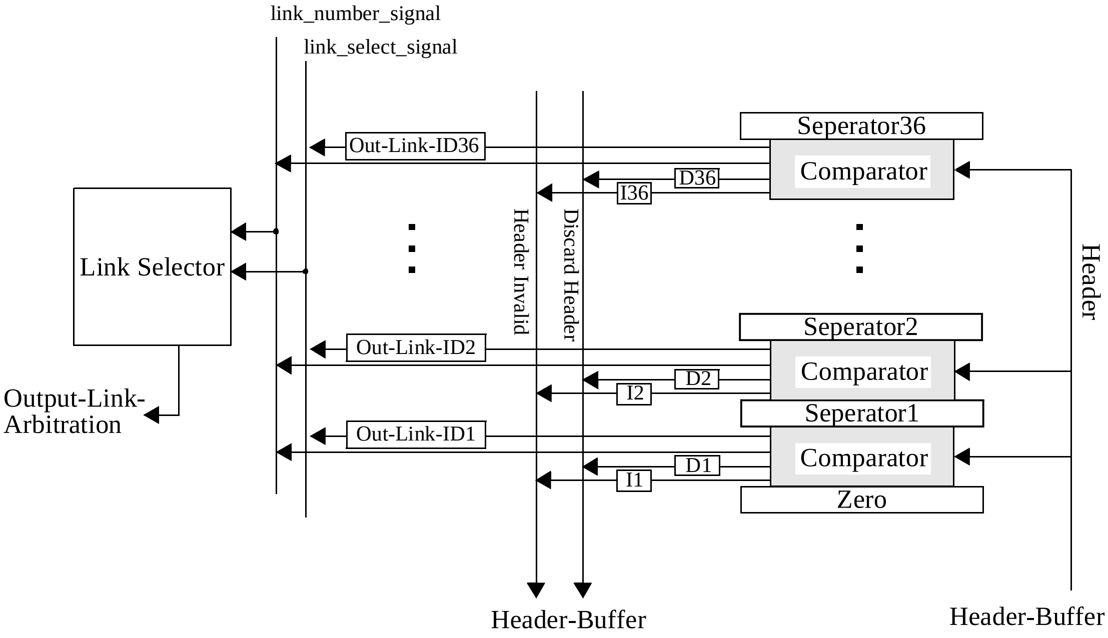


Abb. 3-6: Struktur des Interval-Selectors

Sobald der *Interval-Selector* den Ausgangslink bestimmt hat und der *Crossbar* die Verbindung zu diesem aufgebaut hat, werden die Header- und Datentoken des aktuellen Pakets über den *Crossbar* zum *Header-Stripper* des Ausgangslinks weitergeleitet. Dort wird bei entsprechender Konfiguration evtl. ein Headerbyte entfernt. Dies dient dazu, hierarchische Netze zu implementieren (vergl. [MaTW 93]). Vom *Header-Stripper* gehen dann die Daten zum *Data-Out-Manager* des Ausgangslinkmoduls, der sie unter Berücksichtigung der Flußkontrolle in ein DS-Link Signal umwandelt und nach außen weiterleitet.

3.1.3 Aspekte der Modellierung des STC104 in VHDL

Bei der Modellierung wurde der gesamte Switch auf Gatterebene unter ausschließlicher Verwendung der in Tabelle 3-1 angegebenen Komponenten implementiert. Einzige Ausnahme hiervon ist der *Command Processor*, der komplett als Verhaltensmodell vorliegt, da dieser nur während der Konfiguration des Switches, d.h. zur Belegung der Konfigurationsregister und zum Starten der Links verwendet wird. Die Konfigurationsregister sind Bestandteil des jeweiligen Links und somit wieder als Gatter implementiert. Da die Fehlerauswirkungen im Switch nur während der Kommunikationsphase, d.h. nach der kurzen Initialisierung und Konfiguration untersucht werden sollen, ist eine Gatterbeschreibung des *Command Processors* nicht notwendig. Desweiteren wurde darauf verzichtet, die *Random-Header* Einheit zu implementieren, da dies nur im Rahmen einer Modellierung eines komplexen Netzes aus mehreren Switches sinnvoll wäre.

Um komplexe Verhaltensweisen im Switch zu modellieren, wurden der *Controller* und der *Data-Out-Manager* des *Link-Moduls* sowie der *Header-Buffer* und der *Header-Stripper* jedes Links in Form von endlichen Automaten implementiert [Sti 97]. Diese Verhaltensbeschreibungen wurden dann mit Hilfe des Synthesetools SYNOPSIS^{TM} in ein Gattermodell umgewandelt, das wiederum ausschließlich die in Tabelle 3-1 angegebenen Komponenten beinhaltet.

Gatter	**Anzahl der Eingänge**			
	1	**2**	**3**	**4**
AND		x	x	x
OR		x	x	x
NAND		x	x	x
NOR		x	x	x
D-FlipFlop			x	
RS-FlipFlop		x		
Inverter	x			
Multiplexer		x		x
XOR		x	x	x
Tri-State Treiber	x			

Tab. 3-1: Verwendete Gattertypen

Es sei in diesem Zusammenhang noch einmal ausdrücklich darauf hingewiesen, daß das in dieser Arbeit erstellte VHDL-Gattermodell des STC104 keinen Anspruch darauf erhebt, identisch mit der Realisierung der auf dem Markt befindlichen Hardware zu sein. Das in dieser Arbeit entwickelte VHDL-Modell stellt nur eine der möglichen Realisierungen dar. Dies genügt zur Untersuchung der Auswirkungen von Fehlern der Gatterebene eines Switches auf das Verhalten des Kommunikationsprotokolls beziehungsweise auf das Recovery-Verhalten von dessen Komponenten.

Gemäß dem Datenblatt des STC104 muß der Switch in der Lage sein, eine Kommunikationsbandbreite von 100 Mbits/s pro Link bei einer Latenzzeit von weniger als 1 μs pro Paket aufzuweisen. Um diese Vorgaben zu erreichen, wurde die Verzögerungszeit für Signale in den Gattern auf 250 Pikosekunden eingestellt und der interne Takt im Modell des STC104 auf 100 MHz gesetzt was dem heutigen Stand der Technik entspricht.

Es zeigte sich jedoch, daß dieses ursprüngliche Modell mit 32 vollständig auf Gatterebene beschriebenen Links zu groß für eine statistische Auswertung ist. Bei 427616 Gattern und über 2.5 Millionen möglichen Stuck-At-Fehlern ist der Speicherbedarf und die Simulationszeit des Modells zu groß, um mehrere tausend Fehlerinjektionsexperimente mit VERIFY durchzuführen[1].

Da die 32 Links des Switches bis auf die Belegung der Konfigurationsregister alle identisch aufgebaut sind, wurden zum Zwecke der Fehleranalyse nur vier der 32 Links implementiert. Da bei einer Beschränkung auf vier Links weniger Routingmöglichkeiten vorhanden sind, wurde die Anzahl der Intervalle beim *Interval-Labeling* entsprechen von 36 auf 6 reduziert. Damit sind wie bei dem Orginalsystem mit 32 Links mehr Intervalle als Ausgangslinks verfügbar, wodurch der Zwang direkt aufeinanderfolgender Intervalle bei der Zuordnung der möglichen Empfänger zu den Ausgangslinks vermieden wird.

Weiterhin ist nur ein Link des Switches vollständig auf Gatterebene modelliert, da man sich bei den Fehlerinjektionsexperimenten auf einen der vier identischen Links und den ebenfalls vollständig auf Gatterebne modellierten Crossbar beschränken kann. Die größte Anzahl an Komponenten und damit den größten Anteil der Simulationszeit ist im *Interval-Selector* und im *FIFO*-Puffer des *Link-Moduls* zu finden. Durch den Aufbau des Modells ist sichergestellt, daß ein Fehler, der in dem vollständig auf Gatterebene modellierten Link injiziert wird, sich nicht in die Subeinheiten des *Interval-Selectors* (bzw. des *FIFO*-Puffers) der anderen Links ausbreiten kann. Zur Beschleunigung der Simulationszeit wurde je ein Verhaltensmodell dieser beiden Subkomponenten erstellt, das ein identisches zeitliches und funktionelles Verhalten wie das entsprechende Gattermodell aufweist. Diese Verhaltensmodelle ersetzen bei drei der vier Links die entsprechenden Gattermodelle[2]. Alle Untereinheiten dieser drei Links, deren Verhalten möglicherweise von einem injizierten Fehler im vollständig auf Gatterebene modellierten Link beeinflußt werden können, sind weiterhin auf Gatterebene modelliert. Damit ist es möglich, die Fehlerausbreitung und ein evtl. erfolgtes Recoveryverhalten des modellierten Switches detailliert zu untersuchen.

---

1. Eine Analyse des tatsächlichen Speicherbedarfs und der Simulationszeit wird in Kapitel 6.1 vorgestellt.
2. Selbst im Falle des Fehlermodells mit Übersprechfehlern ist durch die zu erwartende räumliche Trennung dieser Komponenten keine gegenseitige Einflußnahme möglich.

In Tabelle 3-2 wird zur Übersicht die Anzahl der Komponenten der einzelnen Subeinheiten des für die Messungen verwendeten Modells des STC104-Switches aufgeführt. Die Angaben in den mit *Link* gekennzeichneten Zeilen der Tabelle entsprechen dabei dem vollständig auf Gatterebene modellierten Link.

In den beiden Fußzeilen der Tabelle findet sich eine Gegenüberstellung des ursprünglichen Modelles mit 32 vollständig auf Gatterebene modellierten Links und des verkleinerten, 4 Links umfassenden Modells, das für die Messungen herangezogen worden ist. Durch die oben erwähnten Maßnahmen konnte demnach eine Verkleinerung des Modells um circa den Faktor 40 erreicht werden.

			Anzahl der Gatter	**Anzahl der möglichen Stuck-At-Fehler**	**Anzahl der möglichen Bridge-Fehler**
Crossbar und Crossbar Arbiter			400	2416	1720
Link	Header-Buffer		364	2489	539
	Header-Stripper		47	297	91
	Interval-Selector		2072	11546	5215
	Link-Module	Controller	221	1550	453
		Data-Out-Manager	181	1230	248
		Shift-Register	29	205	50
		Data-Valid-Analyser	7	54	11
		FIFO-Puffer	1042	7956	4368
	Link-Modul Gesamt		1380	10995	5130
	Konfigurationsregister		480	4320	975
Link Gesamt			4443	29656	12756
STC104 Gesamt		**4 Link Modell**	**9616**	**67568**	**14476**
		32 Links	**427616**	**2538880**	**1104512**

Tab. 3-2: Gatter der Subkomponenten des STC104 und Anzahl möglicher Fehler

3.2 Modellierung eines ATM-Switches

Wie eingangs des Kapitels schon erwähnt wurde, ist beim ATM-Protokoll keine Flußkontrolle auf den unteren Protokollschichten vorgesehen. Im folgenden werden die Grundlagen des ATM-Protokolls vorgestellt und es wird auf die Komponenten des modellierten ATM-Switches eingegangen. Dabei wurden die von der ITU[1] festgelegten Protokollspezifikationen für die physikalische Schicht und die ATM-Schicht des Protokolls zur Modellierung herangezogen [G.703], [I.311], [I.321], [I.361], [I.362] und [I.363]. Für die Modellierung des ATM-Switches wurde weiterhin eine von den ATT Bell Laboratories unter Führung von Yeh und Kollegen [YeHA87] entwickelte Knockout-Architektur verwendet. Eine weitere Übersicht zu ATM ist in [Pry 93] zu finden.

3.2.1 Grundlagen zum ATM-Protokoll

Obwohl es keine offizielle Zuordnung vom ATM-Protokoll zum ISO/OSI[2]-Referenzmodell für Kommunikationssysteme gibt, läßt sich gemäß [Pry 93] für weite Teile des ATM-Protokolls eine Abbildung der Schichten finden. In Abb. 3-7 wird das von der ITU vorgegebene Broadband-ISDN (BISDN) Protokollmodell für ATM vorgestellt. Dieses Modell besteht im wesentlichen aus drei Ebenen: die *Benutzerebene* zum Transport von Benutzerinformationen, die *Kontrollebene* für die Signalisierungsinformation und schließlich die *Managementebene* zur Wartung des Netzwerks. Jede dieser Ebenen ist wiederum als Schichtenmodell gemäß OSI aufgebaut, wobei die verschiedenen Schichten unabhängig voneinander sind. Die verschiedenen Ebenen und Schichten des ATM-Protokolls werden hier nur soweit beschrieben, wie es für die Modellierung des Switches notwendig ist. Dabei sei darauf hingewiesen, daß die Protokolle der Managementebene nicht modelliert wurden, da sie auf die Fehlerinjektion keinen Einfluß haben (s.u.).

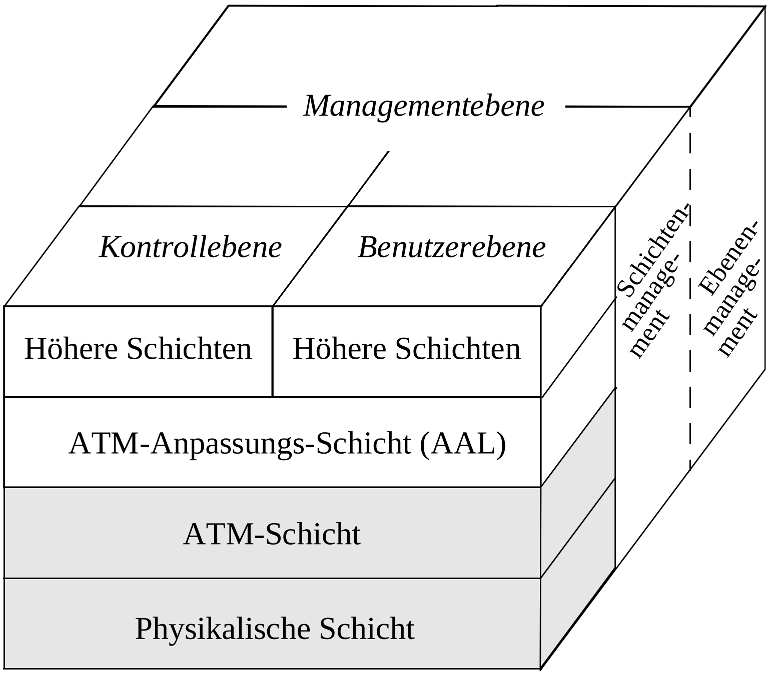


Abb. 3-7: B-ISDN ATM Protokoll-Referenz-Modell

---

1. International Telecommunication Union (ehemals CCITT)
2. International Standard Organisation, Open System Interconnect

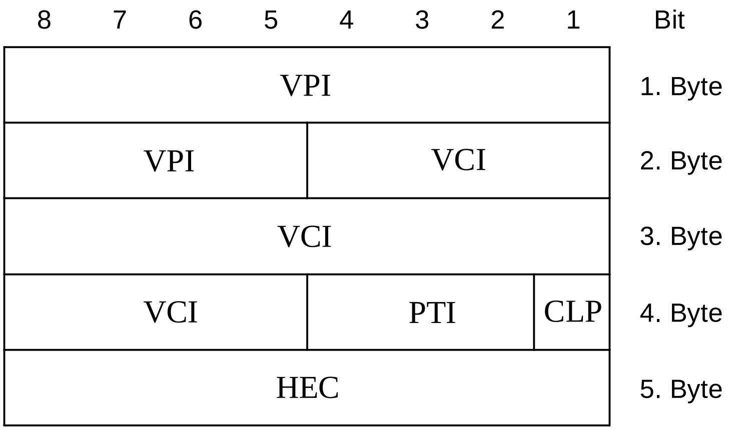
3.2.1.1 Die ATM-Schicht

Die ATM-Schicht entspricht in ihrer Funktionalität der unteren Ebene 2 des OSI-Modells [Pry 93] und ist unabhängig vom physikalischen Medium. Die Hauptaufgaben dieser Schicht bestehen im Multiplexen und Demultiplexen der Zellen verschiedener Verbindungen zu einem einzelnen Strom von ATM-Zellen, der Umsetzung der in der Zelle befindlichen Wegewahlinformationen, verschiedenen (hier nicht berücksichtige) Managementfunktionen und dem Entfernen und Anhängen der Wegewahlinformation zu/von der darüberliegenden AAL-Schicht. Je nachdem wie der Link des Switches konfiguriert ist, existieren zwei verschiedene Formate des Zellheaders: das User Network Interface (UNI) und das Network Node Interface (NNI). Für diese Arbeit wurde das NNI-Format für alle Links eines ATM-Switches verwendet. Im Gegensatz zu Protokollen wie TCP/IP oder dem oben erläuterten STC104-Protokoll ist bei ATM die Größe eines Pakets fest vorgegeben. Aus diesem Grund wird hier von Zellen anstatt von Paketen gesprochen.

Eine ATM-Zelle besteht immer aus einem fünf Byte großen Header und 48 Byte Nutzdaten. Auf der über der ATM-Schicht liegenden AAL-Schicht werden die zu versendenden Nutzdaten mit einem CRC[1] versehen und auf die verschiedenen Zellen aufgeteilt. Der Zellheader kann jedoch nicht von höher liegenden Protokollen geschützt werden, da er erst in der ATM-Schicht erzeugt wird. Aus diesem Grund hat der Zellheader ein acht Bit breites Feld namens HEC (Header Error Control), bei dem ein CRC über die ersten 32 Bit eingetragen wird. Das Generatorpolynom für diesen CRC ist:

$$x^8 + x^2 + x + 1$$

Die ersten 12 Bit des Headers bestehen beim NNI aus dem Virtual Path Identifier (VPI), anschließend kommen 16 Bit des Virtual Channel Identifiers (VCI), ein drei Bit breites Feld, das den Typ der Nutzinformation angibt (PTI: Payload Type Identifier) und abschließend ein CLP (Cell Loss Priority) Bit mit dem Zellen priorisiert werden können. In Abb. 3-8 ist der Aufbau eines NNI-Zellheaders noch einmal graphisch dargestellt.



8	7	6	5	4	3	2	1	Bit
VPI								1. Byte
VPI				VCI				2. Byte
VCI								3. Byte
VCI				PTI			CLP	4. Byte
HEC								5. Byte

Abb. 3-8: Aufbau des Headers einer ATM-Zelle

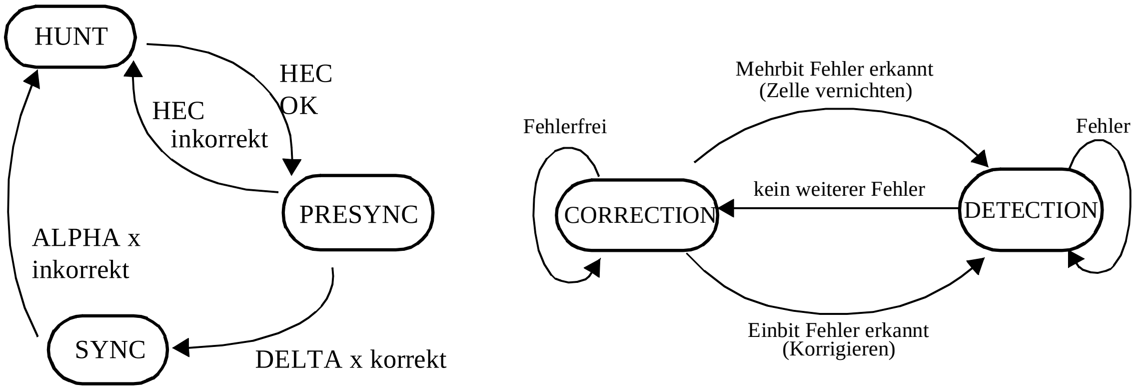
---

[1] CRC: Cyclic Redundancy Check

3.2.1.2 Die Physikalische Schicht

Die physikalische Schicht entspricht hierbei der Ebene 1 des OSI-Modells und erfüllt die Funktionen auf Bitebene. Dabei gibt es drei verschiedene Verfahren, Zellen zu transportieren. Mit PDH[1] und SDH[2] können die Zellen in größere Rahmen eingefügt werden. Dies ermöglicht die Nutzung schon vorhandener Übertragungsnetze. In dieser Arbeit wird jedoch eine zellbasierte Kommunikation verwendet, d.h. daß die Zellen ohne weitere Rahmeninformationen auf der Bitebene übertragen werden. Dieser Teil der physikalischen Schicht ist abhängig vom Übertragungsmedium (optisch, elektrisch, ...) und beinhaltet die korrekte Rekonstruktion des Bit-Zeitverhaltens beim Empfänger.

Daneben beinhaltet diese Ebene auch noch die Transmission Convergence Sublayer (TC-Sublayer). Dieser Teil geht davon aus, daß die Bits schon erkannt worden sind. Die Aufgabe der TC-Sublayer besteht dann darin, aus dem Bitstrom einen ATM-Zellstrom zu erzeugen. Dabei wird der HEC des Headers benutzt, um die Zellgrenzen während eines Kommunikationsaufbaus zwischen zwei Links zu erkennen. In Abb. 3-9a ist dazu das Zustandsdiagramm zur Synchronisation angegeben. Zu Beginn des Kommunikationsaufbaus zwischen zwei Links befindet sich die Erkennungslogik im Zustand *HUNT*. Aus dem eintreffenden Bitstrom wird nach jedem Bit überprüft, ob sich aus den bisher empfangenen Bits ein gültiger HEC ergibt. In diesem Fall wird in den Zustand *PRESYNC* übergegangen. Ab jetzt wird nicht mehr jeder Bitwert des eingehenden Bitstroms auf einen gültigen HEC überprüft, sondern es wird zellsynchron, d.h. nach je *53* Byte (*5* Byte Header und *48* Byte Nutzdaten) eine Überprüfung durchgeführt. Nachdem *DELTA* mal ein gültiger HEC empfangen wurde, geht der Link in den Zustand *SYNC*. Ist jedoch in der Zwischenzeit ein ungültiger HEC eingetroffen, wird davon ausgegangen, daß die bisherigen HEC nur durch Zufall richtig erkannt wurden und der Link geht wieder in den Zustand *HUNT*. Nachdem der Link sich synchronisiert hat (Zustand *SYNC*) wird erst dann wieder davon ausgegangen, daß die Verbindung zum Sender abgerissen ist, wenn *ALPHA* mal nacheinander ein ungültiger HEC empfangen wurde. Die ITU hat für zellbasierte Kommunikation die Werte *ALPHA = 7* und *DELTA = 8* festgelegt.



a) Synchronisationszustände

b) Fehlerverhalten im eingeschwungenen Zustand

Abb. 3-9: ATM-Verhalten bei Header-Fehlern

---

[1] Plesiochronous Digital Hierarchy [G.703]

[2] Synchronous Digital Hierarchy [G.708]

Nachdem sich der Link auf den Zellstrom synchronisiert hat (Zustand *SYNC*) wird der HEC zur Erkennung und Korrektur von Bitfehlern im Header verwendet. Das in Kapitel 3.2.1.2 vorgestellte Generatorpolynom für den HEC ermöglicht eine Korrektur von Einbitfehlern und eine Erkennung von Mehrbitfehlern. Das Verhalten bei Headerfehlern im *SYNC*-Zustand wird in Abb. 3-9b vorgestellt. Wird ein Mehrbitfehler erkannt, wird die aktuelle Zelle verworfen und vom Zustand *CORRECTION* zum Zustand *DETECTION* übergegangen. Nach dem ersten Auftreten eines Einbitfehlers wird dieser korrigiert und ebenfalls in den *DETECTION* Zustand übergegangen. Alle weiteren Fehler, die im Zustand *DETECTION* auftreten, führen zu einer Verwerfung der jeweiligen Zelle. Erst nachdem wieder ein fehlerfreier Header empfangen wurde, geht der Link wieder in den Zustand *CORRECTION*.

3.2.2 Der Strukturelle Aufbau eines Knockout-Switches

Die Modellierung des Switches erfolgte gemäß den Vorgaben der von Yeh und Kollegen an den ATT Bell Laboratories entwickelten Knockout-Architektur. Dabei werden die Zellpuffer, die durch ihre hohen Anforderungen an die Geschwindigkeit den teuersten Teil des Switches darstellen, an den Ausgangslinks plaziert. In Abb. 3-10 ist der Grobaufbau eines Knockout-Switches aufgezeichnet. Jeder der N Eingangslinks wird über ein Kabel mit Daten versorgt, wobei die Daten auf den Leitungen gemäß ITU-Recommendation [G.703] CMI (Coded Mark Inversion) kodiert sind. Anschließend folgen Komponenten, die die oben erwähnten Funktionen der ATM-Schicht und der physikalischen Schicht implementieren. Nachdem der Header einer empfangenen ATM-Zelle ausgewertet und der Ausgangslink bestimmt wurde, muß der aktuellen Zelle ein log_2N breiter interner Header mit der Nummer des Ausgangslinks angehängt werden. Zusätzlich hat man ein Aktivitätsbit, das angibt, ob die aktuelle Zelle Nutzinformation trägt, oder ob es sich um eine eingefügte Idlezelle handelt. Das Businterface jedes Ausgangslinks erkennt anhand des internen Headers, ob die aktuell auf dem Bus liegende Zelle an den zugehörigen Ausgangslink weitergeleitet werden muß und puffert gegebenenfalls die jeweilige Zelle. Bevor die weiterzuleitende Zelle ausgegeben werden kann, muß der interne Header und das Aktivitätsbit wieder entfernt und der CRC für den Zellheader neu berechnet werden. Dies ist notwendig, da in der ATM-Schicht eventuell neue VPI bzw. VCI Felder erzeugt wurden. Abschließend wird der Zellstrom dann wieder CMI-kodiert und nach außen weitergeleitet.

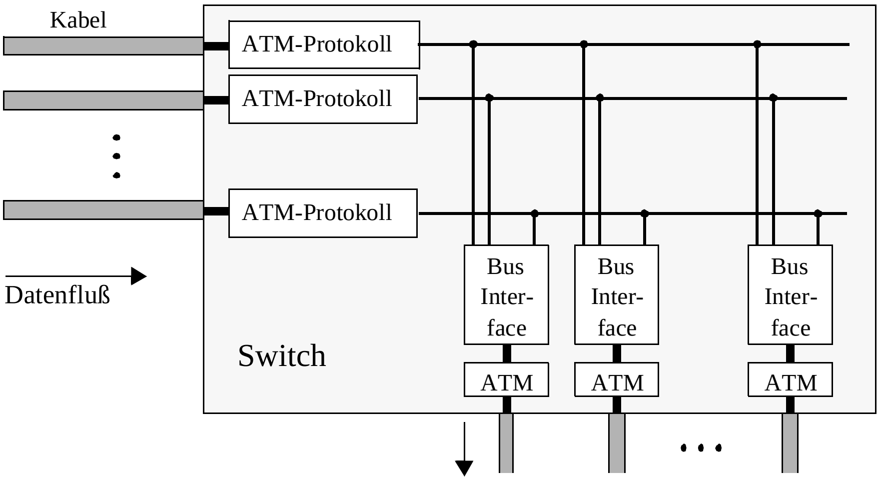


Abb. 3-10: Grobaufbau des Knockout-ATM-Switches

Im folgenden werden die Funktionalität und die Komponenten des Businterfaces genauer beschrieben, da dies der Ort der Fehlerinjektion sein wird. Jedes Businterface hat N Eingänge, die von den Bussen der Eingangslinks des Switches gespeist werden. Mit je einem zugeordneten Zellfilter wird anhand des internen Headers und des Aktivitätsbits erkannt, ob es sich um eine gültige Zelle für den zugeordneten Ausgangslink handelt. Um die Anzahl der Puffer möglichst klein zu halten, wird von der Konzentratorkomponente die Anzahl der eingehenden Zellen auf maximal L reduziert. Über einen Shifter werden die Zellen dann auf L Puffer verteilt und zyklisch von jedem Puffer jeweils eine Zelle zum Ausgang weitergeleitet (siehe Abb. 3-11).

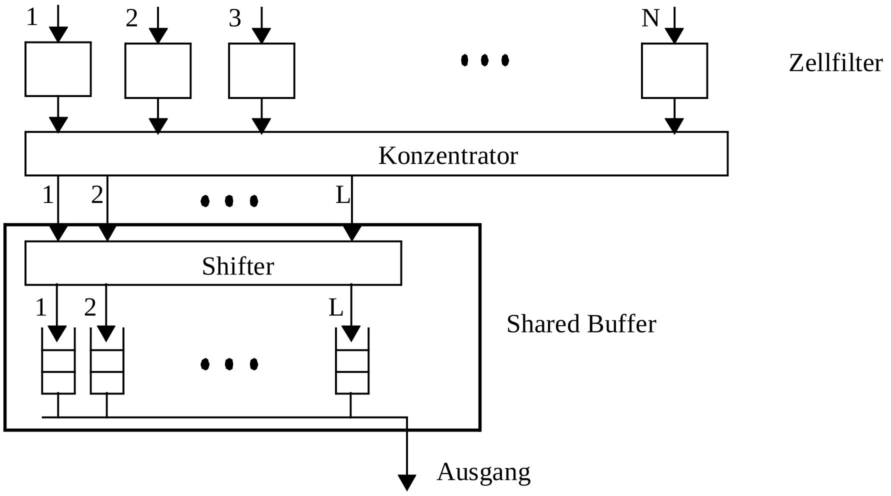


Abb. 3-11: Das Businterface des Knockout-Switches

Wie eingangs des Kapitels schon erwähnt wurde, besteht ein Merkmal einer ATM-Verbindung darin, daß keine Flußkontrolle auf den unteren Ebenen des ATM-Protokolls vorgesehen ist. Damit verbunden ist die Konsequenz, daß Zellen verloren gehen können, falls zu viele Eingangslinks ihre Zellen auf denselben Ausgangslink weiterleiten. In der Knockout-Architektur kann dies entweder beim Konzentrator oder bei einem Überlauf der Puffer geschehen. Der Konzentrator des Businterfaces ist so aufgebaut, daß von $k > L$ gleichzeitig ankommenden Zellen genau L Zellen an den Shifter weitergegeben werden und $k - L$ verworfen werden. Für $k \leq L$ werden alle Zellen weitergegeben.

Diese mögliche Quelle des Zellverlusts macht jedoch nur Sinn, wenn die Wahrscheinlichkeit des Zellverlusts im Konzentrator nicht zu groß ist. Mit der Last p sei die Wahrscheinlichkeit angegeben $(0 \leq p \leq 1)$, mit der eine Zelle an einem gegebenen Eingangslink eintrifft. Unter der Annahme, daß die Zellen unabhängig an den Eingängen des Switches (mit der Last p) eintreffen und gleichverteilt auf die Ausgänge verteilt werden, ist die Wahrscheinlichkeit P_k daß k Zellen gleichzeitig am Konzentrator eintreffen binomial verteilt (vergl. auch [Pry 93]) mit

$$P_k = \binom{N}{k}\left(\frac{p}{N}\right)^k \cdot \left(1 - \frac{p}{N}\right)^{N-k} \qquad k = 0, 1, \ldots, N$$

Wenn nur L Zellen gleichzeitig den Konzentrator passieren können, ergibt sich die Wahrscheinlichkeit eines Zellverlusts gemäß:

$$P[Zellverlust] = \frac{1}{p} \sum_{k=L+1}^{N} (k-L)\left(\binom{N}{k}\left(\frac{p}{N}\right)^{k} \cdot \left(1-\frac{p}{N}\right)^{N-k}\right)$$

In Abb. 3-12 ist für $p=0.9$ die Wahrscheinlichkeit eines Zellverlusts in Abhängigkeit von L für verschiedene Parameter N aufgeführt. Die für ATM typische Anforderung für die Wahrscheinlichkeit eines Zellverlusts von kleiner als 10^{-10} wird also schon für 12 Konzentratorausgänge selbst bei vielen Switcheingängen erreicht.

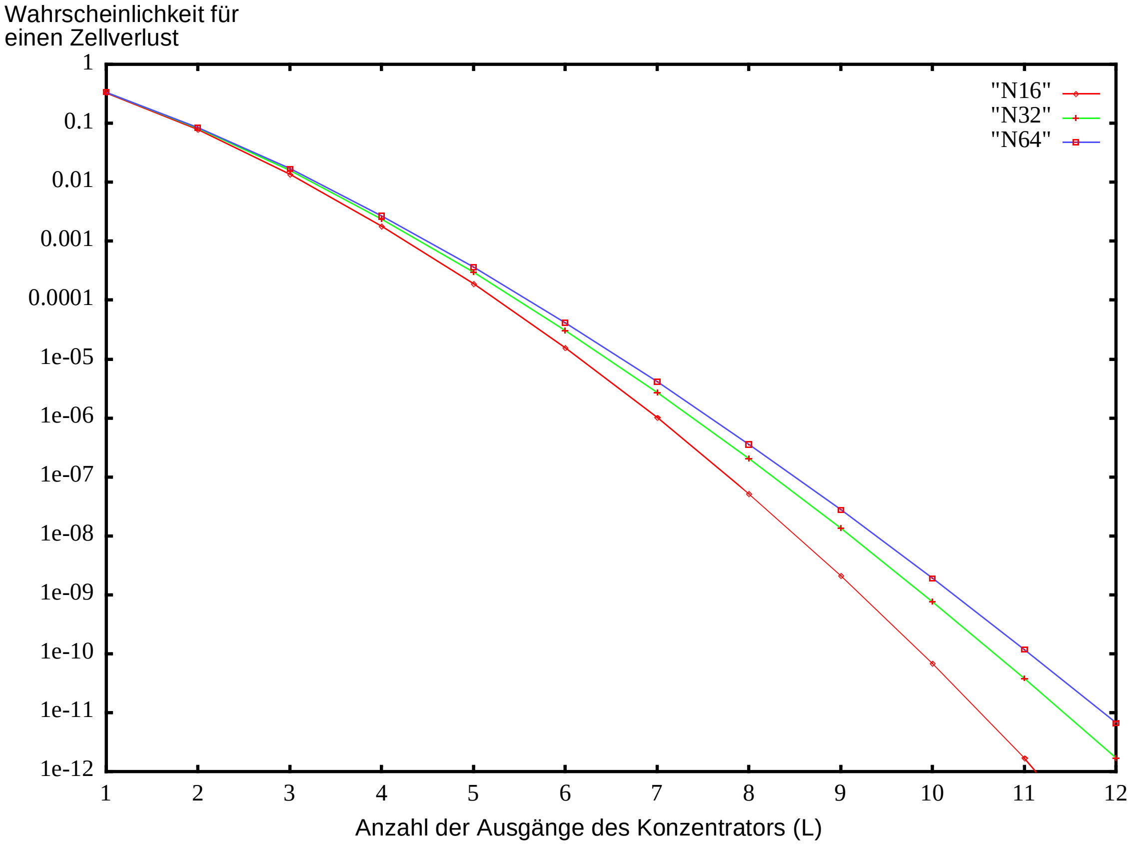


Abb. 3-12: Wahrscheinlichkeit eines Zellverlusts in Abhängigkeit von L bei einer Last von p=0.9

Da der Konzentrator sehr schnell sein muß und mit möglichst wenig Hardwareaufwand implementiert werden sollte, ist er aus sehr einfachen Basiskomponenten aufgebaut. In Abb. 3-13 ist der Aufbau eines Konzentrators mit acht Eingängen und vier Ausgängen dargestellt. Die beiden Hauptkomponenten sind ein Verzögerungselement und ein 2x2-Switch. Falls bei einem 2x2-Switch nur an einem Eingang eine aktive Zelle eintrifft, wird sie direkt an den *winner*-Ausgang weitergeleitet. Treffen zwei Zellen gleichzeitig ein, so wird die Zelle, die am linken Eingang ankommt zum *winner*-Ausgang weitergeleitet, und die rechte Zelle zum *loser*-

Ausgang. Da ein 2x2-Switch mit 6 Gattern implementiert werden kann, ist eine schnelle und kostengünstige Implementierung des Konzentrators gewährleistet.

Durch die in Abb. 3-13 angegebene Verschaltung dieser beiden Basiskomponenten können Konzentratoren mit mehreren Ein- und Ausgängen erzeugt werden. Dabei werden die eintreffenden Zellen durch mehrere Auswahlrunden geschleußt. “Verliert” eine Zelle in der ersten Runde, muß sie sich in der zweiten Runde (d.h. an der zweiten Stufe der 2x2-Switches) qualifizieren. Ein weiterer Vorteil dieser Konzentratorarchitektur ist, daß die Zellen an den Ausgängen des Konzentrators immer “linksbündig” ankommen. Es ist also z.B. nicht möglich, daß an den Ausgängen 1,3 und 4 eine aktive Zelle anliegt, während am Ausgang 2 keine aktive Zelle anliegt. Dies erleichtert es dem Shifter des Businterfaces die Zellen gleichmäßig auf die FIFO-Puffer zu verteilen.

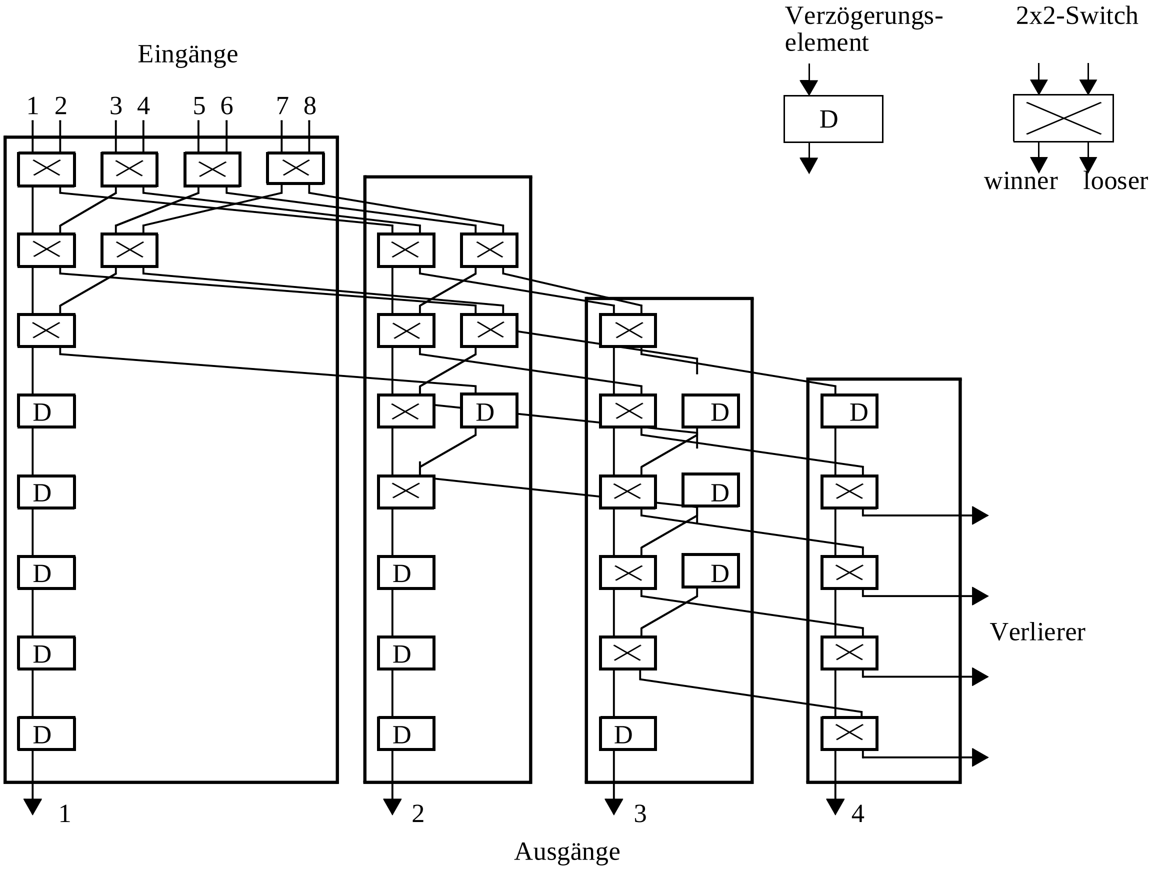


Abb. 3-13: Beispiel eines 8-zu-4 Konzentrators des Knockout-Switches

3.2.3 Aspekte der Modellierung des Knockout-Switches in VHDL

Da über den Aufbau der Komponenten zur Zellsynchronisation und des Routing keine Information vorhanden waren und eine komplette Entwicklung eines Gattermodells dieser Komponenten den Rahmen dieser Arbeit gesprengt hätten, wurden nur die Komponenten des Businterfaces als Gattermodell implementiert. Dies hat den Vorteil, daß die Ergebnisse der

Fehlerinjektion unabhängig von der Betriebsart des Switches (SDH, PDH oder Zellorientiert) sind. Alle Komponenten außerhalb des Businterfaces sind aus diesem Grund als Verhaltensmodell in VHDL implementiert. Wie beim Modell des STC104 sei auch hier noch einmal darauf hingewiesen, daß die gewählte Realisierung in Gattern nur eine der möglichen Realisierungen darstellt und keinen Anspruch darauf erhebt, identisch mit einer realen Hardware zu sein.

Für die Komponenten des Businterfaces wurde die gleiche Gatterbibliothek verwendet, wie für die Modellierung des STC104 (siehe Tabelle 3-1). Wie beim STC104 mußte auch hier die Anzahl der Links auf vier reduziert werden, um eine statistische Auswertung in vertretbarer Zeit zu erreichen. Der Konzentrator ist aus diesem Grund mit den Parametern $N = 4$ und $L = 2$ ausgelegt. Die jeweils zwei FIFO-Puffer der Businterfaces haben eine Kapazität von je fünf Zellen. In Tabelle 3-3 ist die Anzahl der Gatter pro Businterface angegeben. Der ATM-Switch ist so spezifiziert, daß er gemäß ITU Recommendation [G.703] eine Datenübertragung von *155520* Kbits/s pro Link verarbeiten kann. Der interne Takt vor dem Anhängen der internen Addressbits und des Aktivitätsbits beträgt ca. *155* MHz, da die Daten bitseriell übertragen werden. Da das Verhältnis von internen Bits zu den Zellbits sehr gering ist, ergibt sich in den Businterfaces mit den zusätzlichen internen Bits ein nur unwesentlich höherer Takt. Die Verzögerungszeiten der Gatter sind in der gleichen Größenordnung wie beim Modell des STC104.

Neben der Reduzierung des Switches auf vier Links mußte die Größe der ATM-Zellen und damit die Anzahl der Gatter in den FIFO-Puffern verkleinert werden, um eine vertretbare Simulationszeit zu erreichen. Die in dem Modell verwendeten ATM-Zellen bestehen aus einem Byte Nutzdaten, während der Header unverändert genau dem NNI-Spezifikationen entspricht. In Tabelle 3-3 wird für die Komponenten des Businterfaces die Anzahl der Gatter und der möglichen Stuck-At-Fehler aufgeführt. Neben den Angaben, die für die Modellierung verwendet wurden, sind die Gatterzahlen auch für den FIFO-Puffer mit den Parametern $N = 16$, $L = 8$, der eigentliche ATM-Zellgröße von *53* Byte und einer FIFO-Pufferkapazität von je 5 Zellen angegeben. Diese Parameter entsprechen einer Implementierung mit einer Gesamtwahrscheinlichkeit eines Zellverlusts von 10^{-8}.

		Anzahl der Gatter	**Anzahl der möglichen Stuck-At-Fehler**
Verwendetes Modell mit N=4, L=2 und 1 Byte Nutzdaten	Zellfilter	140	896
	Konzentrator	31	254
	Shifter	10	71
	FIFO-Puffer	1984	15614
	Businterface Gesamt	2170	16835
Realistisches Modell mit N=16, L=8 und 48 Byte Nutzdaten	Zellfilter	140	896
	Konzentrator	630	5210
	Shifter	< 100	< 900
	FIFO-Puffer	98168	757176
	Businterface Gesamt	~ 99000	~ 764000

Tab. 3-3: Gatter der Subkomponenten des Businterfaces und Anzahl möglicher Fehler

Die Größenordnung bezüglich der Anzahl der Gatter und Anzahl möglicher Stuck-At-Fehler ist vergleichbar mit dem Modell des STC104. Gegenüber dem ATM-Switch mit *16* Links, *8* Puffern pro Link, einer Zellgröße von *53* Byte und *5* Zellen pro Puffer ist die Größe des Modells durch die Annahmen um den Faktor *45* verringert worden. Bridging-Fehler wurden in dem Modell des ATM-Switches nicht berücksichtigt da dies den Rahmen der Arbeit gesprengt hätte.

3.3 Lastmodelle

Die Auswirkungen eines Fehlers im Switch hängen wesentlich davon ab, ob die betroffenen Einheiten gerade an einer Kommunikation beteiligt sind oder ob sie sich in einem sogenannten Idle-Zustand befinden. Es ist also notwendig, diese Lastabhängigkeit so detailliert wie möglich bei den Fehlerinjektionsexperimenten zu berücksichtigen. Ein erster Schritt hierzu war die hardwarenahe Modellierung der Switches. So wird zum Beispiel der *Interval-Selector* des STC104 nur für die Bytes des Headers benötigt. Fehler, die in dessen Subeinheiten injiziert werden, während der Link gerade kein Paket weiterzuleiten hat, wirken sich demgemäß auch nicht auf das End-zu-End Protokoll aus.

Neben der hardwarenahen Modellierung des Switches muß ein adäquates Lastmodell der Kommunikation vorhanden sein. Die im Rahmen dieser Arbeit entwickelten Lastmodelle stellen den Switches Daten gemäß des jeweiligen Kommunikationsprotokolls zur Verfügung.

3.3.1 Das Lastmodell des STC104

Wie in Kapitel 3.1.3 schon angesprochen wurde, mußte das Modell des STC104 auf 4 Links begrenzt werden, um eine akzeptable Simulationszeit zu erreichen. Da es sich, wie schon erwähnt, in dieser Arbeit nicht um eine exakte Modellierung des von INMOS hergestellten Switches handeln kann, wurde auch der Lastgenerator so entworfen, daß die Simulationszeit für mehrere tausend Fehlerinjektionen im Rahmen dieser Arbeit zu bewältigen war. Dazu wurde die maximale Paketgröße auf *5* Byte Nutzdaten und einem Byte für den Header reduziert. Im Gegensatz zum Systemmodell des ATM-Switches wirkt sich beim STC104 die Paketgröße nicht auf die Hardware aus, da diese so konzipiert ist, daß sie Pakete variabler Größe verarbeiten kann. Der Inhalt der Nutzdaten wurde dabei, ebenso wie das Headerbyte, von einem Zufallsgenerator erzeugt.

Jeweils eine Instanz des Lastgeneratormoduls ist mit einem der Links des STC104 verbunden. Die Instanzen dieser Module sind so parametrisierbar, daß sie Datenpakete für mehrere mögliche Empfänger generieren können. Nach der Initialisierungssequenz, die beim Aufbau der Linkverbindung notwendig ist, läuft die Datengenerierung nach dem in Abb. 3-14 angegebenen Schema ab.

Die Grundlage dieses Algorithmus ist das ON-OFF Modell eines Kommunikationsverhaltens. Hierbei wechseln jeweils aktive und passive Phasen der Kommunikation ab. Nachdem die Dauer der aktive Phase, d.h. des Nachrichtenaustausches mit Hilfe eines Zufallswertes festgelegt wurde, wird eine Menge von Nachrichten ermittelt, die in dieser Phase versendet werden. Während der aktiven Phase wird jeweils ein Datenpaket aus der Menge der Nachrichten bestimmt und anschließend an den Header, je nach verbleibender Restlänge der Nachricht, maximal fünf Byte Nutzdaten versendet. Je nachdem, ob alle Bytes der jeweiligen Nachricht versendet wurden, wird das Paket mit einem End-Of-Message-Token (EOM) oder einem End-Of-Packet-Token (EOP) abgeschlossen. Im Anschluß an die aktive Phase, folgt dann eine wiederum zufällig ausgewählte Zeitdauer, bei der keine Nutzdaten versendet werden. Während beider Phasen werden Flußkontroll- bzw. NUL-Tokens versendet, so daß das gesamte Protokollspektrum des STC104 abgedeckt ist.

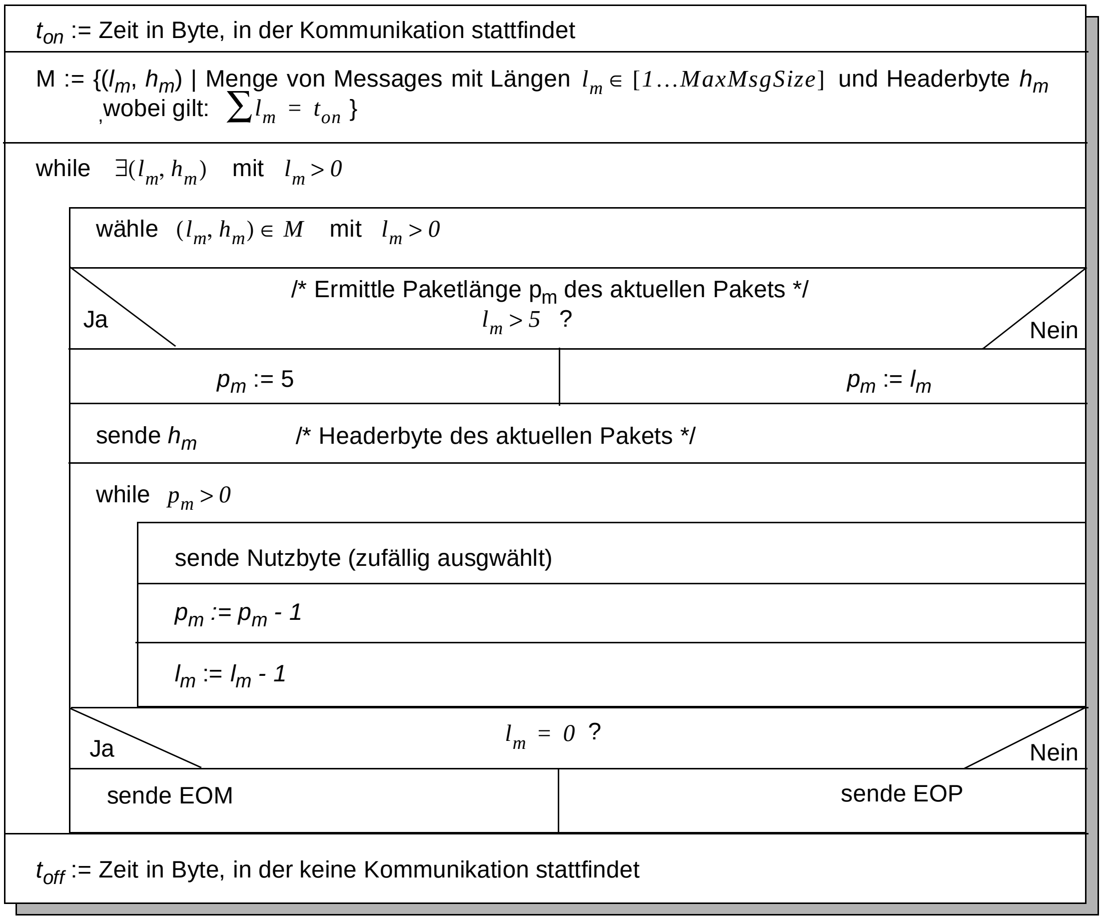


Abb. 3-14: Algorithmus zur Datengenerierung beim STC104 Lastmodell

Jeder der vier Lastgeneratoren protokolliert die gesendeten und empfangenden Token der Nutzdaten und die EOP- bzw. EOM-Token in einer Datei. Dies ermöglicht eine Analyse der Auswirkungen der injizierten Fehler auf das Protokollverhalten des STC104. Desweiteren generieren die Lastgeneratoren die NUL- bzw. die FCT-Token, die der STC104 benötigt und wandelt die generierten Bits der Token in DS-Linksignale um.

3.3.2 Das Lastmodell des Knockout-Switches

Um die Auswirkungen eines Hardwarefehlers auf verschiedene Kommunikationsanforderungen zu untersuchen, wurde im Rahmen dieser Arbeit ein parametrisierbarer Lastgenerator für ATM entwickelt. Damit lassen sich ATM-Lastquellen verschiedener Dienstklassen, d.h. gemäß gegebener QoS-Parameter (**Q**uality **o**f **S**ervice), implementieren. In Tabelle 3-4 werden die durch ATM definierten Dienstklassen gemäß ihrer Charakteristika dargestellt.

Klassen	**A**	**B**	**C**	**D**
Timing	erforderlich		nicht erforderlich	
Bitrate	deterministisch	stochastisch		
Verbindungsart	verbindungsorientiert			nicht verbindungsorientiert
Beispiel	**DBR-Video**	**SBR-Video**	**FTP**	**Broadcast**

Tab. 3-4: Die ATM Dienstklassen und ihre Charakteristika

Aus der Tabelle werden die drei Hauptkriterien zur Unterscheidung der Dienstklassen ersichtlich:

1. *Timing zwischen Sender und Empfänger*: Mit diesem Kriterium wird festgelegt, ob zwischen Sender und Empfänger eine feste zeitliche Beziehung besteht (wie zum Beispiel bei der ISDN-Telefonie).

2. *Bitrate*: Hierbei wird unterschieden, ob die Bitrate während der gesamten Übertragungsdauer konstant (deterministisch) oder variierend (stochastisch) ist. Im ersten Fall spricht man von DBR-Diensten (**D**eterministic **B**it **R**ate), im zweiten von SBR-Diensten (**S**tochastic **B**it **R**ate).

3. *Verbindungsart*: Ein Dienst, der mehr als einen Empfänger hat, wird als “verbindungslos” gekennzeichnet. Hat der Dienst nur einen Empfänger, wird er als “verbindungsorientiert” bezeichnet.

Jede der vier angegebenen Dienstklassen hat ihre Entsprechung in einer Ausprägung der AAL-Schicht (**A**TM-**A**daption-**L**ayer) des ATM-Protokols. Diese vier Dienstklassen lassen sich jedoch noch feiner unterteilen, wenn man die Anforderungen berücksichtigt, die sich aus der Funktion des Dienstes ergeben. Bei der Bitrate kann dabei zusätzlich die mittlere bzw. die maximale Bitrate zur Unterscheidung berücksichtigt werden. Bei der *Kommunikationsqualität* wird festgelegt, wie groß die maximal tolerierbare Zellverlustrate sein darf und wie groß die Verzögerungen sein dürfen, die während der Kommunikation erlaubt sind. Dabei spricht man von einem *Jitter* zur Beschreibung der erlaubten Verzögerungsabweichung bei aufeinanderfolgenden ATM-Zellen. Betrachtet man z.B. den Sprachdienst genauer, so kann man die Kommunikation in eine *Talkspurt*- und in eine *Silence*-phase unterteilen. Dabei werden die Pausen zwischen zwei aufeinanderfolgenden Sätzen, Worten, usw. berücksichtigt. Die *Silence*-phase entspricht einer Sprechpause, während der keine Daten übertragen werden müssen. Mit der ADPCM-Kodierung[1] kann dabei im DSI-Verfahren[2] die Zwischenankunftszeit nahezu verdoppelt werden (von 6 *ms* auf 12 *ms*).

Die bei dem in dieser Arbeit entstandenen Lastgenerator verwendeten QoS-Parameter sind:

- *Größe der zu versendenden Daten.* Diese wird durch eine Verteilungsfunktion bestimmt, mit der die Anzahl der zu versendenden Bytes während der aktiven Phase des Dienstes ermittelt wird. Diese Verteilung ist abhängig davon, ob es sich um einen DBR- oder um einen SBR-Dienst handelt. Nimmt man den FTP-Dienst als Beispiel, so ist damit die Uniformverteilung für diese aktive Phase festgelegt.
- *Die Zwischenankunftszeit*: Diese Zeit wird durch eine weitere Verteilungsfunktion bestimmt, und gibt an, wie lange die jeweils passive Phase des Dienstes ist, in der keine Daten erzeugt werden. Auch diese Verteilungsfunktion ist abhängig von der Klasse des Dienstes (DBR oder SBR). Beim FTP-Dienst ist die Dauer dieser Phase z.B. exponentiell verteilt.

Die Tabelle 3-5 gibt einen Überblick der Parameter verschiedener ATM-Dienste, die mit dem entwickelten Lastgenerator simuliert werden können.

Bezeichnung	**Dienstklasse**	**Datenmenge (DM) [Bytes]**	**Zwischenankunftszeit (ZA) [ms]**	**Verteilungen (DM/ZA)**
DBR_VOICE-PCM	ICO	48	6	DET/DET
AUDIO_ON_DEMAND	ICO	48	0.25	DET/DET
DBR_VOICE_ADPMC	ICO	48	12	DET/DET
SBR_VIDEOPHONE	SCO	200-45K	33.3	UNIF/EXP
SBR_GRAPHIC	SCO	6M	5000	DET/EXP
SBR_FTP3	ACO	1500-25K	35.333	UNIF/EXP
SBR_FTP10	ACO	100-1M	400.04	UNIF/EXP

Tab. 3-5: Beispiele simulierbarer ATM-Lastquellen und ihre Parameter

Die Abkürzungen DET, EXP und UNIF stehen dabei für eine deterministische, exponentielle und uniforme Verteilung der Datenmengen bzw. der Zwischenankunftszeiten. Die Dienstklassen unterscheiden sich nach isochron-verbindungsorientiert (ICO), synchron-verbindungsorientiert (SCO) und asynchron-verbindungsorientiert (ACO). Die Parameter des Dienstes SBR_GRAPHIC entsprechen einer Übermittlung digitaler Röntgenbilder mit *2000x2000* Pixeln und *12*-Bit Graustufen. Beim SBR_FTP3 handelt es sich um einen File-Tranfer mit einer mittleren Bitrate von *3* MBits/s während der SBR_FTP10 Dienst einen File-Transfer mit durchschnittlich *10* MBits/s darstellt.

---

[1] **A**daptive **D**ifferential **P**ulse **C**ode **M**odulation (für Details sei auf das ITU-Dokument [G.726] verwiesen)

[2] **D**igital **S**peech **I**nterpolation [Bra 68]

Wie schon beim Systemmodell des ATM-Switches mußten auch bei der Modellierung des Dienstes Anpassungen vorgenommen werden, um vertretbare Simulationszeiten zu erhalten. Die Beobachtungszeit des Systems nach Injektion eines Stuck-At-Fehlers im Businterface wurde auf *2500* ns beschränkt. Zusammen mit der Verkleinerung der ATM-Zellen entspricht das einer Anzahl von durchschnittlich acht Zellen, die in dieser Zeit vom Switch empfangen und weitergesendet werden können.

KAPITEL
4

Quantitative Analyse des Ausfallverhaltens

Dieses Kapitel beschäftigt sich mit der Auswertung der Fehlerinjektionsexperimente auf der Gatterebene der beiden betrachteten Systeme. Im Mittelpunkt der Untersuchungen stehen dabei die Recovery-Zeiten der Systeme nach einem Fehler und die Verteilung der Fehler, die zu einem dauerhaften Ausfall des Systems (Crash) führen. Unter der *Recovery-Zeit* wird in diesem Zusammenhang die Zeitdauer verstanden, die zwischen der Injektion des Fehlers und der selbständigen Rückkehr des Systems in einen fehlerfreien Zustand vergeht. Es wird bei dieser Untersuchung also explizit davon ausgegangen, daß es keine externen Fehlerbehebungsmaßnahmen während des untersuchten Zeitraums gibt.

Bei den Untersuchungen des STC104 Switches wurden dabei neben den Messungen mit dem Stuck-At-Fehlermodell auch Fehlerinjektionen auf Basis von Übersprechfehlern (siehe Abschnitt 2.3.1 auf Seite 16) durchgeführt. Die damit verbundenen Unterschiede des Recovery-Verhaltens des STC104 werden dargestellt.

4.1 Erläuterung der Diagrammdarstellung

Die Untersuchung des Recovery-Verhaltens der Systeme wird analog zu der für VERIFY vorgeschlagenen Diagrammform dargestellt [Sie 98]. Dieses Unterkapitel ist der Erläuterung dieser Darstellungsform gewidmet und soll die Interpretation der Ergebnisse vereinfachen. Die Gesamtzeit eines Systems, in das ein Fehler injiziert wird, kann in maximal vier Phasen aufgeteilt werden:

- Ph_0, die Zeit beginnend vom Start des Systems bis hin zum Beginn der Aktivierung des Fehlers
- Ph_1, die Dauer der Aktivierung dieses Fehlers

- Ph_2, die Zeit zwischen Ende der Aktivierung bis hin zum Verschwinden der Auswirkungen des Fehlers im beobachteten System
- Ph_3, Phase nachdem der Fehler gänzlich aus dem beobachteten System verschwunden ist

Hat man es mit reaktiven Systemen zu tun, ist im Allgemeinen ein Zeitlimit T_{max} vorgeben, bis zu dem das System eine Antwort liefern muß, um die Bearbeitung der Aufgabenstellung des Systems zu gewährleisten. Somit kann es also in zwei Fällen zu einem Wegfall der Phase Ph_3 kommen: falls T_{max} noch während der Phase Ph_2 erreicht wird oder falls sich das System gar nicht mehr erholen kann (z.B. wenn ein Konfigurationsregister durch den Fehler einen falschen Wert erhält).

In Abb. 4-1 soll nun an einem Beispiel die Aussage einer fiktiven Meßkurve veranschaulicht werden. Auf der X-Achse des Diagramms ist dazu die Zeit beginnend mit dem Start der Phase Ph_2 aufgetragen. Zum Zeitpunkt $t=0$ ist der Fehler also nicht mehr aktiv. Die Werte der Y-Achse bezeichnen hierbei die Wahrscheinlichkeit $P(t)$, daß die Auswirkungen eines Fehlers zur Zeit t aus dem beobachteten System verschwunden sind.

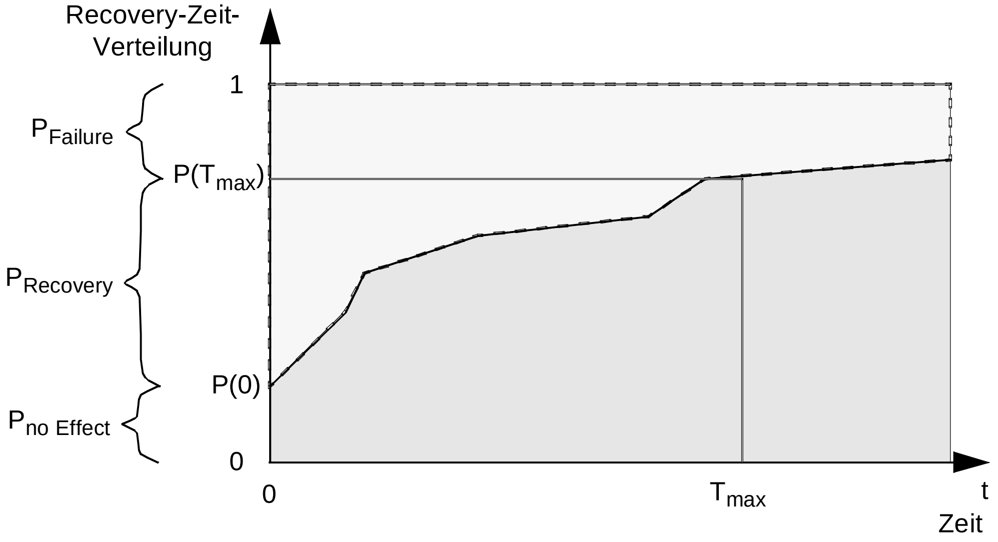


Abb. 4-1: Beispieldiagramm

$P(T_{max})$ ist die Wahrscheinlichkeit für das Verschwinden der Auswirkungen des Fehlers zwischen Ende der Aktivierung und T_{max}. Mit $P(0)$ ist die Wahrscheinlichkeit angegeben, mit der ein Fehler direkt nach Ende der Injektionsdauer (d.h zum Zeitpunkt $T = 0$) nicht mehr beobachtbar ist und damit keinerlei Auswirkungen auf das weitere Systemverhalten hat[1].

Mit dieser Darstellung der Meßergebnisse läßt sich nun eine Reihe weiterer Aussagen über das Systemverhalten machen. Es ist leicht ersichtlich, daß mit dem Wert $P(0)$ (im folgenden mit $P_{no\ Effect}$ gekennzeichnet) die Wahrscheinlichkeit angegeben ist, daß ein aufgetretener Fehler im System sofort verschwindet und keine Auswirkungen nach sich zieht. Als weitere wichtige

---

[1] Beispiel eines solchen Fehlers wäre ein Stuck-At-0 Fehler an einem Eingang eines UND-Gatters während der andere Eingang auch mit dem Wert ’0’ gespeist wird.

Systemgröße ist mit $P_{Recovery} = P(T_{max}) - P(0)$ die Wahrscheinlichkeit einer erfolgreichen Recovery-Aktion ausgedrückt, die ohne äußere Fehlerbehebungsmaßnamen ausgekommen ist[1].

$P_{Failure} = 1 - P(T_{max})$ ist schließlich die Wahrscheinlichkeit, mit der sich das beobachtete System von einem Fehler nicht mehr rechtzeitig erholt, d.h. daß der Fehler entweder überhaupt nicht behebbar ist, oder daß der Fehler nicht in der vorgegebenen maximalen Zeitspanne behoben werden konnte.

In einem zweiten Schritt soll nun das Diagramm so weit verfeinert werden, daß sich Informationen über die einzelnen Subkomponenten daraus ablesen lassen. Dazu sei mit $K = \{k_1, \ldots, k_n\}$ die Menge aller Subkomponenten des Gesamtsystems bezeichnet und mit $A_i \;\; i \in \{i \ldots n\}$ das Ereignis, daß ein Fehler in die Subkomponente k_i injiziert wird. Gemäß der bei VERIFY geltenden Einzelfehlerannahme sind die Ereignisse A_i alle paarweise disjunkt (siehe dazu Kapitel 2.2.2 auf Seite 13). Für das zufällige Ereignis B_t, daß die Auswirkungen eines in das Gesamtsystem injizierten Fehlers in einer Zeit kleiner oder gleich t behoben ist gilt somit: $B_t = (A_1 \cap B_t) \cup \ldots \cup (A_n \cap B_t)$. Mit $P_k(t) = P(A_i \cap B)$, d.h. der Wahrscheinlichkeit, daß ein Fehler, der während eines Fehlerinjektionsexperiments in der Subkomponente $k \in K$ aktiviert wurde, in weniger als t Zeiteinheiten aus dem Gesamtsystem verschwunden ist, folgt nun:

$$P(t) = P(B_t) = \sum_{i=1}^{n} (B_t \cap A_i) = \sum_{k \in K} P_k(t)$$

Zur graphischen Veranschaulichung dieses Sachverhalts wird der dunkelgrau unterlegte Bereich der Kurve der Abb. 4-1 unterteilt. Zur Interpretation der daraus resultierenden Diagrammform, ist in Abb. 4-2 ein Beispiel einer Recovery-Zeitverteilung eines Systems gegeben, das aus den drei Komponenten *A*, *B* und *C* bestehe.

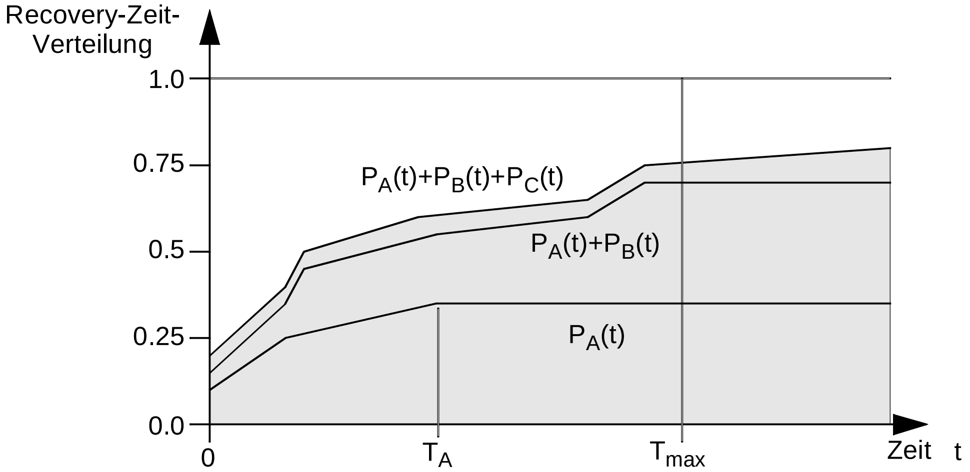


Abb. 4-2: Unterteilung der Recovery-Zeiten bereits behobener Fehler

---

[1] Beispiel einer solchen Recovery-Aktion wäre ein falscher Signalwert, der als Eingangswert an einem ODER-Gatter anliegt, an dessen anderem Eingang eine ’1’ anliegt. Es handelt sich hierbei also meistens um eine Maskierung von Fehlern.

Aus der Abb. 4-2 wird ersichtlich, daß Fehler in der Komponente *A* nur bis zum Zeitpunkt T_A behoben werden. Fehler, die erst nach diesem Zeitpunkt aus dem System verschwinden, sind ausschließlich in den beiden anderen Komponenten *B* und *C* aufgetreten. Da in dem Beispiel die oberen beiden Kurven bis zum Zeitpunkt T_{max} annähernd parallel verlaufen, kann geschlossen werden, daß Fehler in der Komponente *C* überhaupt nicht aus dem System verschwinden. Fehler der Komponente *B* sind für diejenigen Recovery-Aktionen verantwortlich, die länger als T_A Zeiteinheiten dauern. Diese Diagrammform erlaubt es also anzugeben, wann die Fehler der einzelnen Subkomponenten behoben werden.

Eine weitere wichtige Größe im Zusammenhang mit der quantitativen Analyse des Ausfallverhaltens ist die Information, wieviele Fehler sich noch zu einem gegebenen Zeitpunkt im System befinden, die noch nicht behobenen sind. Diese Information kann dargestellt werden, indem der hellgrau unterlegte Bereich der Abb. 4-1 weiter unterteilt wird. Die Werte der Y-Achse bezeichnen hierbei jetzt die Wahrscheinlichkeit $Q(t)$, die besagt, daß die Auswirkungen eines aufgetretenen Fehlers innerhalb des Zeitraums 0 bis t nicht behoben worden sind. Unterteilt man das Gesamtsystem nun wieder in Subkomponenten, erhält man analog zu den obigen Betrachtungen:

$$Q(t) = \sum_{k \in K} Q_k(t) .$$

Dabei gilt $Q(t) = 1 - P(t)$ und für alle Komponenten $k \in K$: $P_k(t) + Q_k(t) = const.$

Abb. 4-3 zeigt ein Beispiel für eine Unterteilung nach Auftrittsorten bisher noch nicht korrigierter Fehler. Dabei wird der Bereich oberhalb der Kurve für $P(t)$ gemäß den Subkomponenten in einzelne Gruppen unterteilt, wobei man an den dadurch entstehenden Kurvenverläufen die Verteilung der im System verbliebenen Fehler der Subkomponenten ablesen kann. Dabei entsprechen die Werte zum Zeitpunkt $T = 0$ der Häufigkeit der noch zu korrigierenden Fehler in den Subkomponenten. In dem Diagramm ist zu erkennen, daß *30%* dieser Fehler in Komponente *A* auftreten $(Q_A(0) \approx 0{,}3)$, die Komponente *B* circa *35%* aller zu korrigierenden Fehler beherbergt $(Q_B(0) \approx 0{,}35)$ und sich in Komponente *C* etwa *15%* dieser Fehler ereignen $(Q_C(0) \approx 0{,}15)$. Die mit $P(0)$ angegebenen restlichen *20%* der aufgetretenen Fehler hatten, wie oben schon erwähnt, keinerlei Auswirkungen im System.

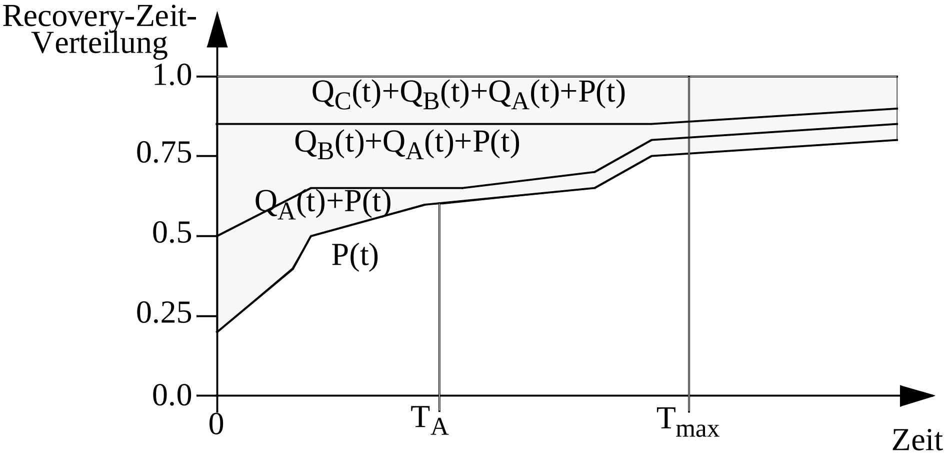


Abb. 4-3: Unterteilung der Recovery-Zeiten noch zu korrigierender Fehler

Die Dynamik dieser Größe in diesem Beispiel läßt sich wie folgt beschreiben. Wie schon in Abb. 4-2 dargestellt, ist die Fehlerkorrektur bei Fehlern in Komponente *A* bis zum Zeitpunkt T_A abgeschlossen. In Abb. 4-3 ist dieser Sachverhalt daran zu erkennen, daß ab diesem Zeitpunkt die Kurven von *P(t)* und von $Q_A(t) + P(t)$ parallel verlaufen. Der Wert von $Q_A(t)$ ist ab dem Zeitpunkt T_A konstant bei *5%*, was besagt, daß *5%* der in Komponente *A* aufgetretenen Fehler nicht beseitigt werden können. Betrachtet man nun den Zeitpunkt T_{max}, so sinkt auch für Komponente *B* der Wert der im System verbleibenden Fehler auf circa *5%*. Bis zu diesem Zeitpunkt werden jedoch keinerlei Fehler der Komponente *C* behoben, was sich aus dem konstanten Wert von $Q_C(t)$ ergibt.

4.2 Recovery-Verhalten des STC104

Im folgenden werden die Ergebnisse der Fehlerinjektionsexperimente am STC104 vorgestellt. Bei den injizierten Fehlern handelt sich ausschließlich um transiente Fehler, die sich aus den Fehlerbeschreibungen der Gatter aus Tabelle 3-1 auf Seite 28 ergeben. Für jeden Eingang und für jeden Ausgang von allen dort angegebenen Gattern ist ein Stuck-At-0 und ein Stuck-At-1 Fehlerverhalten spezifiziert, wobei für jeden Stuck-At Fehler dieselbe mittlere Auftrittshäufigkeit angegeben ist. Bei den Übersprechfehlern ist, wie bei den Stuck-At Fehlern, die mittlere Auftrittshäufigkeit für jeden Fehler gleich. Damit ist in beiden Fehlermodellen gewährleistet, daß jeder mögliche Fehler mit derselben Wahrscheinlichkeit injiziert wird. Die Dauer der Fehler ist gemäß VERIFY exponentiell verteilt, wobei für die in dieser Arbeit durchgeführten Experimente eine mittlere Fehlerdauer von *25 ns* festgelegt wurde.

Gemäß den Betrachtungen über den Experimentumfang in Kapitel 2.2.3 auf Seite 15 wurden für die Untersuchung des Recovery-Verhaltens je *5000* Fehlerinjektionen pro Fehlermodell durchgeführt. Die Beobachtungsdauer jedes Fehlers betrug dabei *500 ns*.

4.2.1 Stuck-At-Fehler

Auf den folgenden beiden Diagrammen (Abb. 4-4 und Abb. 4-5) sind die Recovery-Zeiten bereits behobener Stuck-At Fehler beim STC104 aufgetragen. Abb. 4-4 zeigt dabei den Zeitraum zwischen *0 ns* und *5 ns*. Wie zu erkennen ist, haben die Subkomponenten *Interval Selector* und *FIFO* direkt nach Beendigung der Fehlerinjektion (d.h. zum Zeitpunkt *T=0*) den größten Anteil daran, daß der injizierte Fehler keine Auswirkungen nach sich zieht. Dies ist unter anderem darauf zurückzuführen, daß diese beiden Komponenten mit *36.0%* und *24.8%* der möglichen Fehlerquellen im System die größte Wahrscheinlichkeit haben, eine zufällig ausgewählte Fehlerquelle zu beinhalten[1]. Als Beispielszenario eines solchen Falles kann man sich ein AND-Gatter vorstellen, an dessen erstem Eingang eine ‘0’ anliegt, und durch die Fehlerinjektion der zweite Eingang mit einer ‘1’ statt einer ‘0’ gespeist wird. Der Ausgang dieses AND-Gatters wird sich also durch die Fehlerinjektion nicht ändern.

---

[1] Es sei in diesem Zusammenhang noch einmal darauf hingewiesen, daß alle Fehlerquellen im System dieselbe Fehlerauftrittswahrscheinlichkeit haben.

Die Recovery des Systems wird während der ersten *5 ns* maßgeblich vom *Interval Selector* beeinflußt. Hiebei kann festgestellt werden, daß dies vornehmlich innerhalb der ersten Nanosekunde, d.h. innerhalb von vier Gatterlaufzeiten nach Beendigung der Fehlerinjektion geschieht[1]. Die restlichen Komponenten tragen nicht oder nur unwesentlich zur Recovery des STC104 bei, wie sich an der parallelen Linienführung der Meßergebnisse dieser Komponenten ablesen läßt. Dies läßt sich damit erklären, daß der *Interval Selector* trotz seiner vielen Gatter nur sehr wenige Register enthält. Damit wächst die Wahrscheinlichkeit, daß ein fehlerhaftes Signal beim Durchgang durch mehrere logische Verknüpfungsgatter maskiert wird (Bsp. s.o.). Bei Komponenten mit einem größeren Anteil an Registern ist die Wahrscheinlichkeit auch größer, daß sich ein fehlerhaftes Signal in einem Register manifestiert und somit längere Zeit im System verweilt.

In Abb. 4-5 ist der weitere Verlauf des Recovery-Verhaltens der Komponenten des STC104 aufgezeichnet. Hierbei ist zu erkennen, daß der *Interval Selector* nach diesen ersten *5 ns* nicht mehr weiter zur Recovery des Systems beiträgt. Die Recovery des Systems im weiteren Zeitverlauf wird hierbei hauptsächlich von der *FIFO* mit einer Maskierung von weiteren *5%* der injizierten Fehler bestimmt. Die Gesamtdauer (T_{max}) der Beobachtung des Systems nach dem Ende der Fehlerinjektion wurde mit *500 ns* so gewählt, daß sie über der maximalen Aufenthaltsdauer eines Daten- oder Protokollbits liegt. Aus diesem Grund kann man bei $P(T_{max})$ die Wahrscheinlichkeit ablesen, daß ein aufgetretener Fehler im System zu keiner Recovery und damit zu einem dauerhaften Systemfehler führt. Unter der Annahme eines Stuck-At Fehlermodells liegt dieser Wert gemäß den Messungen bei *94,6%*.

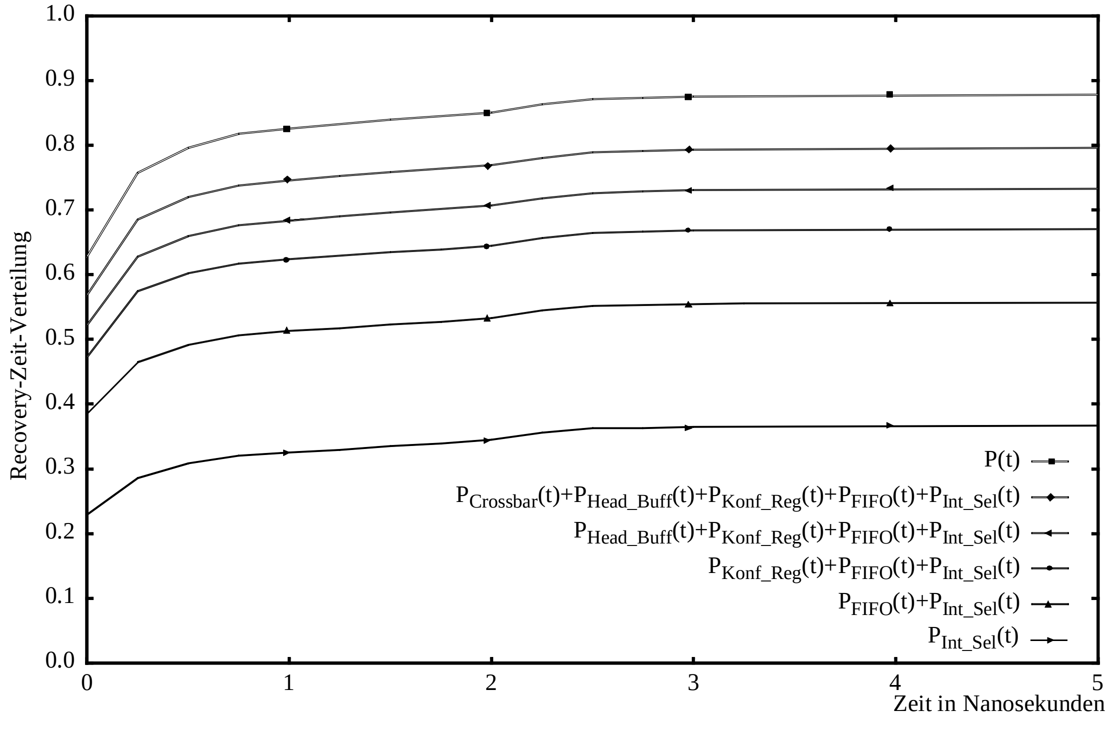


Abb. 4-4: Recovery-Zeiten bereits behobener Stuck-At-Fehler des STC104 (0 - 5ns)

---

[1] Wie in Kapitel 3.1.3 beschrieben, wurden die Schaltverzögerungen der Gatter auf *250 ps* festgelegt.

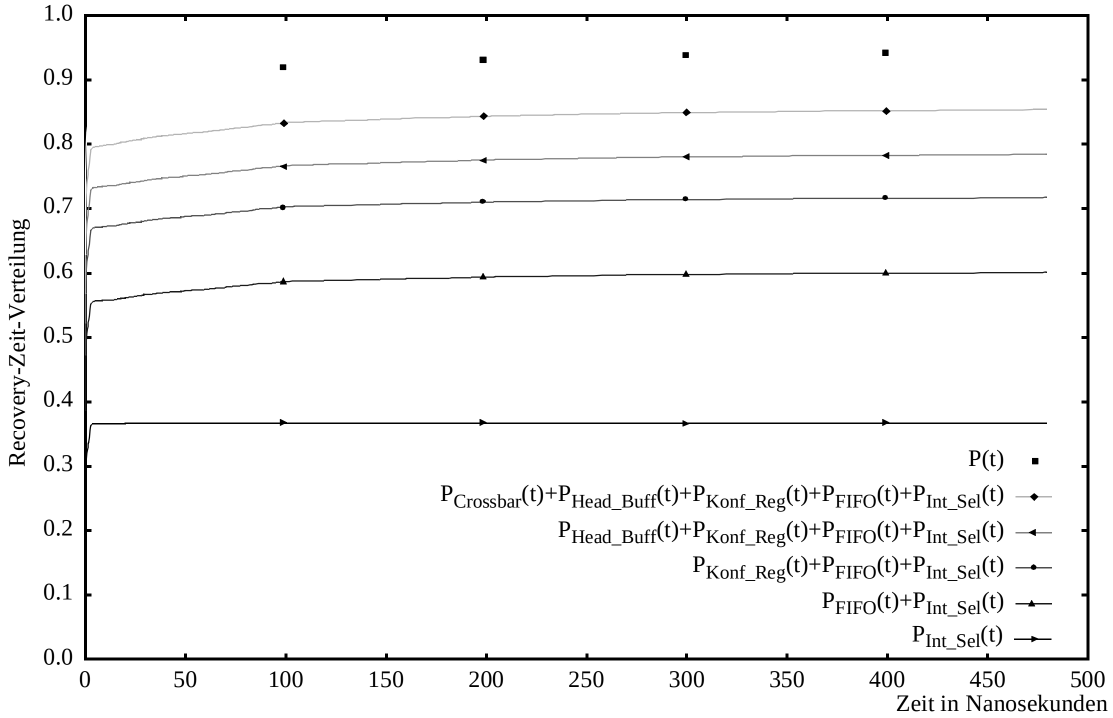


Abb. 4-5: Recovery-Zeiten bereits behobener Stuck-At-Fehler des STC104 (0-500ns)

Damit führen nur *5.4%* aller in das System eingebrachten Stuck-At Fehler zu einem dauerhaften Systemfehler während sich bei *62,9%* gar keine Auswirkungen zeigen und bei *31,7%* eine Recovery stattfindet. Die durchschnittliche Dauer bis zur Recovery wurde mit *23,2 ns* gemessen.

Abb. 4-6 zeigt die Recovery-Zeit Verteilung noch zu behebender Fehler in den jeweiligen Subkomponenten des STC104 während der ersten *5 ns*. Dabei ist hervorzuheben, daß der *Interval Selector* und die *FIFO* mit 15% bzw. 9% die beiden Komponenten sind, die direkt nach Beendigung der Fehlerinjektion den größten Anteil noch zu behebender Fehler besitzen. Während sich jedoch beim *Interval Selector* schon nach *4 ns* fast keinerlei Fehler mehr befinden, die noch zu beheben wären, ändert sich die Anzahl noch zu behebender Fehler bei der *FIFO* in diesem Zeitraum so gut wie gar nicht.

Betrachtet man den weiteren Verlauf der Recovery-Zeit Verteilung bis hin zu $T_{max} = 500$ *ns (Abb. 4-7)*, so erkennt man, daß sich der Anteil noch zu behebender Fehler fast ausschließlich in der *FIFO* verringert. Hierbei ist die stärkste Verringerung bis $T = 100$ *ns* zu verzeichnen. Dies ist dadurch erklärbar, daß in diesem Zeitraum diejenigen Datenbits, die in der FIFO durch einen Fehler verändert wurden, nach und nach den STC104 durch einen Ausgangslink verlassen. Dies ist auch der Grund für die lineare Abnahme der noch zu korrigierenden Fehler.

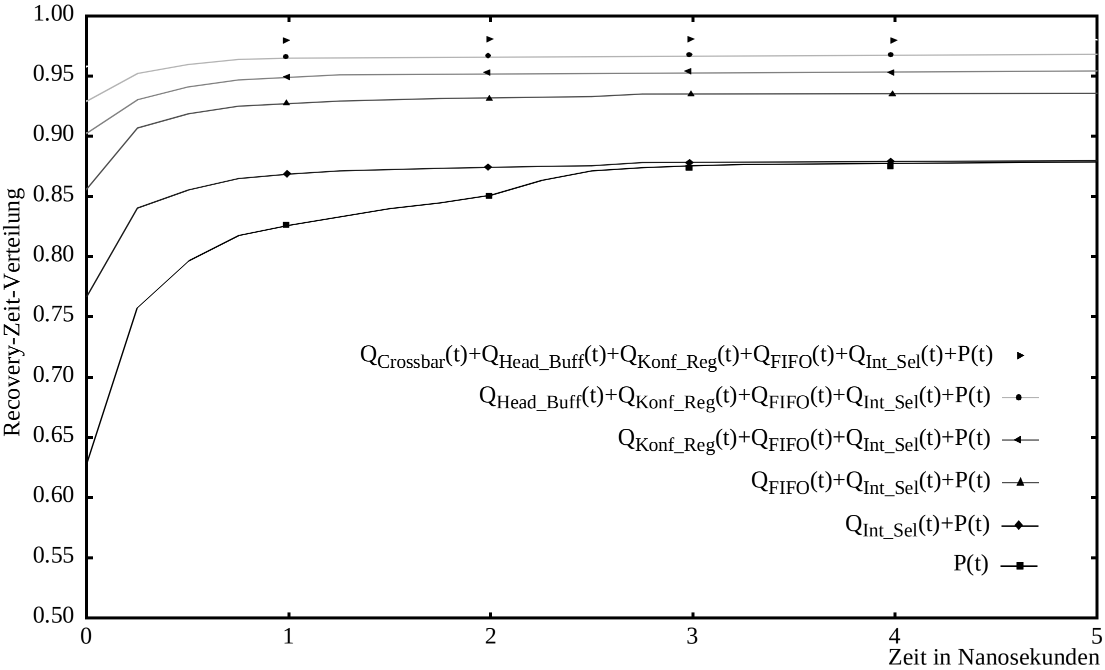


Abb. 4-6: Verteilung der Recovery-Zeiten noch zu behebender Stuck-At-Fehler beim STC104 (0-5ns)

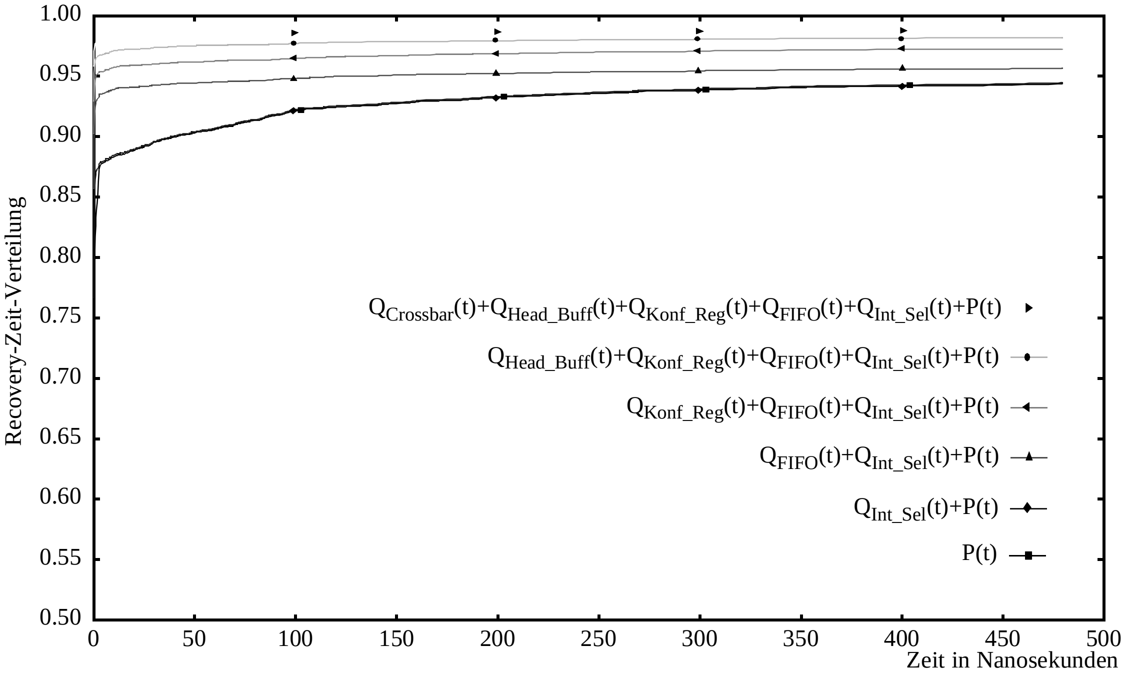


Abb. 4-7: Verteilung der Recovery Zeiten noch zu behebender Stuck-At-Fehler beim STC104 (0-500ns)

An den bisher vorgestellten Meßergebnissen war schon zu erkennen, daß die Anzahl möglicher Fehlerquellen im System einen Einfluß auf das Crash- und Recoveryverhalten der Subkomponenten des STC104 hat. Mit der folgenden Abb. 4-8 soll diese Abhängigkeit genauer untersucht werden. Der mittlere Balken der Abbildung stellt für jede Subkomponente den Anteil möglicher Fehlerquellen im Gesamtsystem dar. Wie oben schon erwähnt, stellt hierbei der *Interval Selector* und die *FIFO* den Hauptanteil, während die Finite State Machines (FSMs)[1] und die *Konfigurationsregister* nur einen geringen Teil der Fehlerquellen beinhalten.

Im linken Balken der Abb. 4-8 ist der Anteil der Komponenten im Falle eines nicht reparablen Systemverhaltens, d.h. einem Crash des STC104 aufgeführt. Hierbei ist zu erkennen, daß der *Interval Selector* trotz des großen Anteils möglicher Fehler (*36,0%*) kaum zu den Crashes des Systems beiträgt (*1,7%*). Im Gegensatz dazu werden die häufigsten Crashes durch Fehler in den *Konfigurationsregistern* hervorgerufen (*29,1%*), obwohl deren Anteil an Fehlerquellen eher gering ist (*13,5%*). Ähnlich zu den *Konfigurationsregistern* haben auch die FSMs eine höhere Wahrscheinlichkeit einen Crash zu verursachen, als es ihr Anteil an der Gesamtzahl möglicher Fehler vermuten läßt.

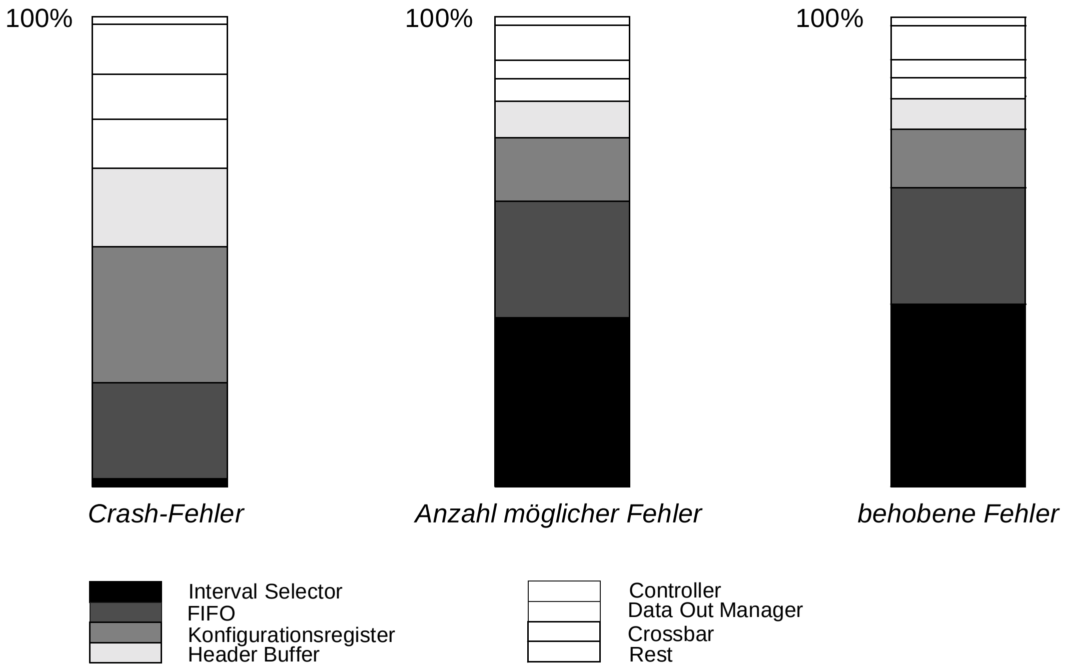


Abb. 4-8: Vergleich der Fehler-, Crash- und Recoverywahrscheinlichkeiten des STC104 bei Stuck-At-Fehlern

Betrachtet man die Komponenten mit höherer Crashwahrscheinlichkeit genauer, so erkennt man, daß es sich hierbei um Systeme mit einem höheren Anteil an Registern handelt, als dies beim *Interval Selector* der Fall ist. Desweiteren läßt sich unterscheiden, ob es sich um Komponenten mit vielen rückgekoppelten Registern (FSMs), Komponenten mit Registern, die zum Datentransfer dienen (*FIFO*) oder um Komponenten mit hauptsächlich zum Lesen verwendeten

---

[1] Bei den FSMs handelt es sich um den *Header Buffer*, den *Data Out Manager*, den *Controller* und den *Header Stripper* wobei letzterer in der Gruppe Rest enthalten ist (vergl. dazu auch Kapitel 3.1.2 auf Seite 24)

Registern handelt (*Konfigurationsregister*). Während der Anteil an Crashes bei Fehlern in der *FIFO* geringer als der Anteil möglicher Fehlerquellen in dieser Komponente ist, wird bei den FSMs ein größerer Anteil verzeichnet. Die größte Steigerung hierbei ist bei dem Registertyp zu erkennen, der hauptsächlich zum Lesen verwendet wird.

Im rechten Balken der Abb. 4-8 ist der Anteil der Komponenten im Falle von den Fehlern aufgeführt, die zu einer Recovery des Systems geführt haben. Vergleicht man diese Werte mit dem jeweiligen Anteil möglicher Fehler (mittlerer Balken), so zeigt sich, daß diese Wertepaare miteinander korreliert sind. Trotz des unterschiedlichen Komponentenaufbaus ist der Anteil behobener Fehler in allen Fällen sehr nah am Anteil aller Fehler.

4.2.2 Übersprechfehler

In den bisherigen Betrachtungen dieses Kaptitels wurden ausschließlich Meßergebnisse aus Fehlerinjektionen mit dem Stuck-At Fehlermodell vorgestellt. Um die Auswirkungen des Fehlermodells auf die Meßergebnisse zu untersuchen, wurden für die Übersprechfehler dieselben Meßmethoden und Auswerteverfahren wie beim Stuck-At Fehlermodell verwendet. Der Unterschied besteht in der ausschließlichen Verwendung von Übersprechfehlern bei der Fehlerinjektion.

In Abb. 4-9 werden die Recovery-Zeiten des STC104 und dessen jeweiligen Komponenten im Zeitraum *0 ns - 5 ns* dargestellt. Ähnlich wie beim Stuck-At Fehlermodell sind auch hier der *Interval Selector* und die *FIFO* die beiden Komponenten mit dem größten Anteil an Fehlern ohne Auswirkungen auf das System. Abweichend zum Stuck-At Fehlermodell hat hier jedoch die *Crossbar* Komponente einen weit gewichtigeren Anteil an den Fehlern ohne Auswirkungen. Dies kann damit erklärt werden, daß im *Crossbar* sehr viele interne Signale (und damit Signalleitungen) vorhanden sind, die direkt zum Datentransport zwischen den Links eingesetzt werden.

Als weitere Beobachtung sei angeführt, daß der Gesamtanteil auswirkungsloser Fehler im System im Falle des Übersprech-Fehlermodells mit *87.0%* deutlich größer ist, als beim Stuck-At Fehlermodell mit *62,8%* (vergl. dazu Abb. 4-9 mit Abb. 4-4). Ein weiterer Unterschied zwischen dem Verhalten der beiden Fehlermodelle zeigt sich im zeitlichen Recovery-Verhalten. Während sich der Gesamtanteil der behobenen Fehler beim Stuck-At Fehlermodell innerhalb der ersten *5 ns* um *25%* auf *87.8%* verbessert hat, ist beim Übersprechfehlermodell nur eine Verbesserung um *7.6%* auf *94.6%* zu verzeichnen.

Die weitere Entwicklung des zeitlichen Recovery-Verhaltens bei Übersprechfehlern ist in Abb. 4-10 für den Zeitraum *0 ns-500 ns* dargestellt. Der in der vorigen Abbildung ersichtliche Trend einer relativ geringen Verbesserung der Recovery über die Zeit wird mit dieser Abbildung bestätigt. Der Gesamtanteil behobener Fehler verbessert sich in der Zeitspanne bis *500 ns* nach Beendigung der Fehlerinjektion noch einmal um *3.6%* auf *98.2%*. Damit hat das Fehlerverhalten unter diesem Fehlermodell eine weitaus geringere Dynamik als beim Stuck-At Fehlermodell. Hierbei ist jedoch bemerkenswert, daß beim Übersprech-Fehlermodell trotz der geringeren Dynamik eine insgesamt bessere Fehlerbehebung stattfindet als beim Stuck-At Fehlermodell. Dies läßt sich mit dem deutlich höheren Anteil auswirkungsloser Fehler begründen.

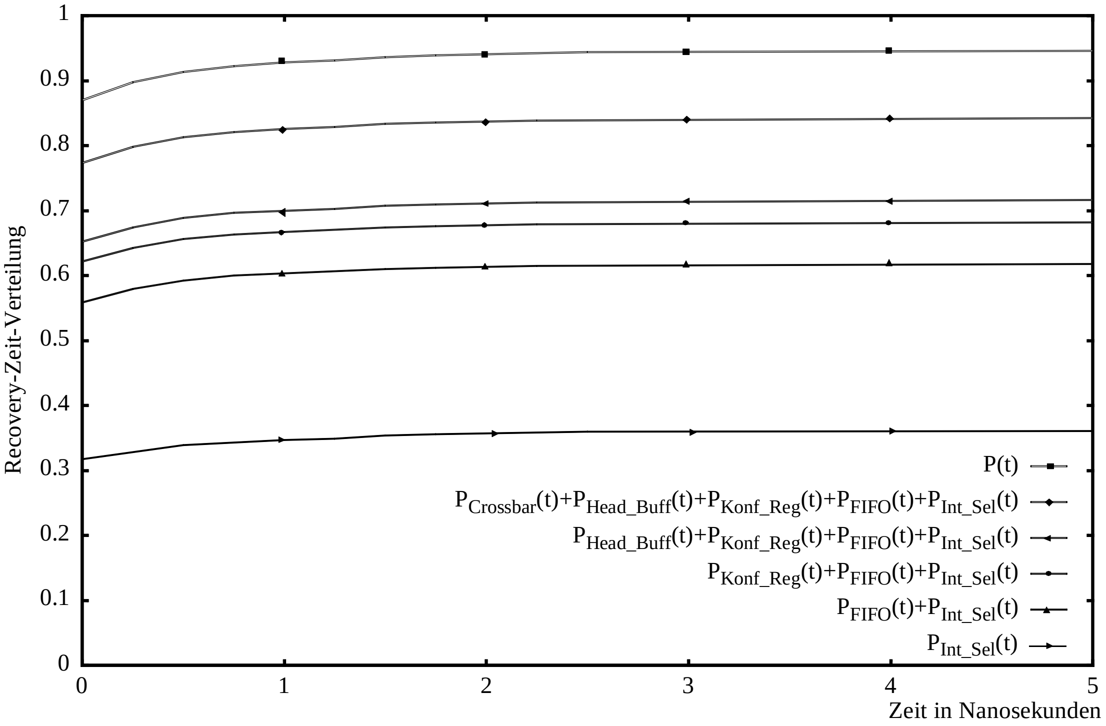


Abb. 4-9: Recovery-Zeiten bereits behobener Übersprechfehler des STC104 (0 - 5ns)

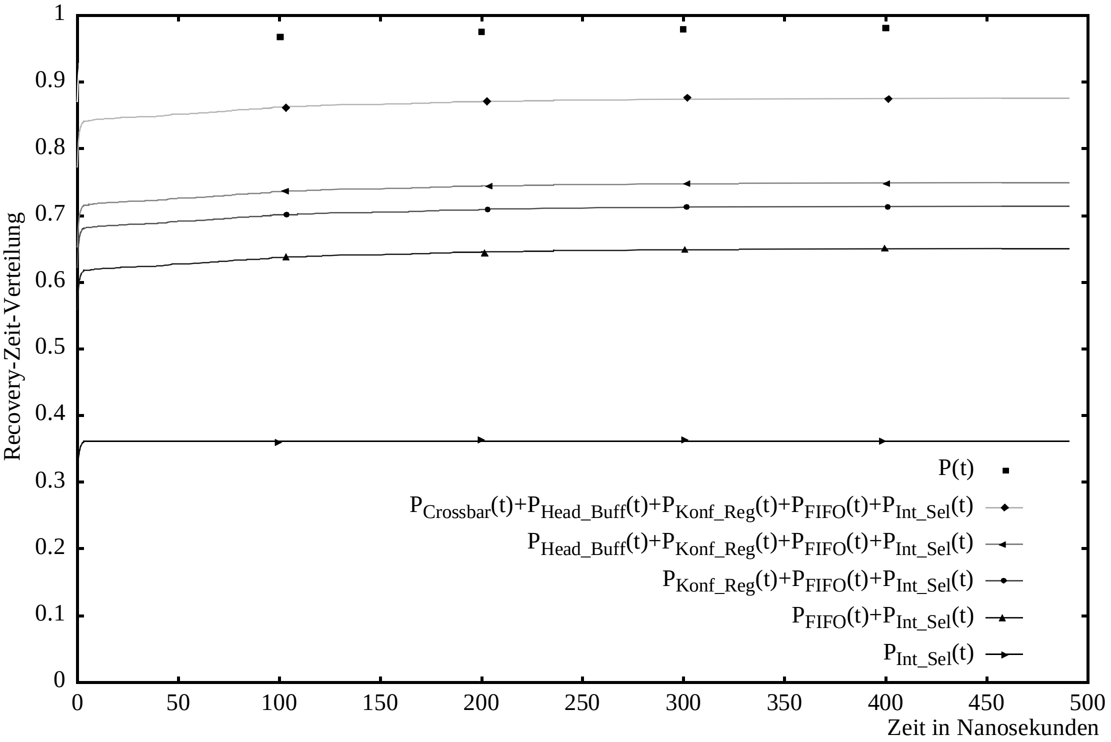


Abb. 4-10: Recovery-Zeiten bereits behobener Übersprechfehler des STC104 (0 - 500ns)

Vergleicht man die Dynamik einzelner Komponenten des Systems bei den verschiedenen Fehlermodellen, so zeigt sich in beiden Fällen ein gleiches Verhalten. Während einer anfänglich erkennbaren Verbesserung beim *Interval Selector* ist in beiden Fällen im Zeitraum zwischen *5 ns* und *500 ns* keine Verbesserung bei dieser Komponente zu erkennen. Demgegenüber ist es genau dieser Zeitraum, in dem in beiden Fällen die *FIFO* eine erkennbare Verbesserung hervorruft.

In Abb. 4-11 und Abb. 4-12 sind die Recovery-Zeiten noch zu behebender Übersprechfehler im STC104 nach Komponenten unterteilt aufgeführt. Hierbei wird zur Vergleichbarkeit der Ergebnisse wieder eine Unterteilung der Zeiträume in *0ns-5ns* (Abb. 4-11) und in *0ns-500ns* (Abb. 4-12) angegeben. Zur besseren Veranschaulichung wurde hier jedoch wegen der geringeren Dynamik ein Wertebereich von *80%* bis *100%* gewählt[1].

Hierbei ist bemerkenswert, daß Übersprechfehler in *Konfigurationsregistern* so gut wie überhaupt nicht zur Recovery des Systems beitragen. Wie beim Stuck-At Fehlermodell verschwindet auch beim Übersprech-Fehlermodell der Anteil noch zu korrigierender Fehler beim *Interval Selector* fast vollständig. Das Verhalten der restlichen Komponenten entspricht im wesentlichen dem des Verhaltens beim Stuck-At Fehlermodell. Das gilt insbesondere auch für das Recovery-Verhalten der *FIFO*, deren Anteil noch zu behebender Fehler sich wie beim Stuck-At Fehlermodell über einen größeren Zeitraum (*250 ns*) kontinuierlich verringert.

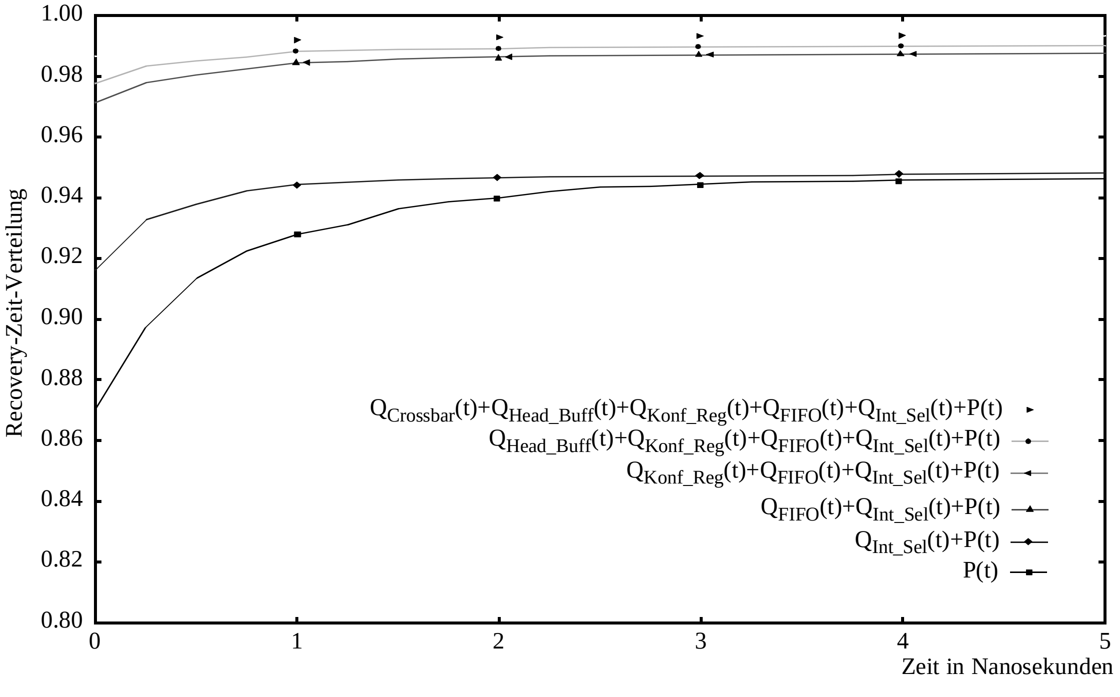


Abb. 4-11: Recovery-Zeiten noch zu behebender Übersprechfehler des STC104 (0-5ns)

---

[1] vergl. dazu Abb. 4-6 und Abb. 4-7: Wertebereich der Ergebnisse noch zu behebender Übersprechfehler beim Stuck-At Fehlermodell beträgt 50% bis 100%.

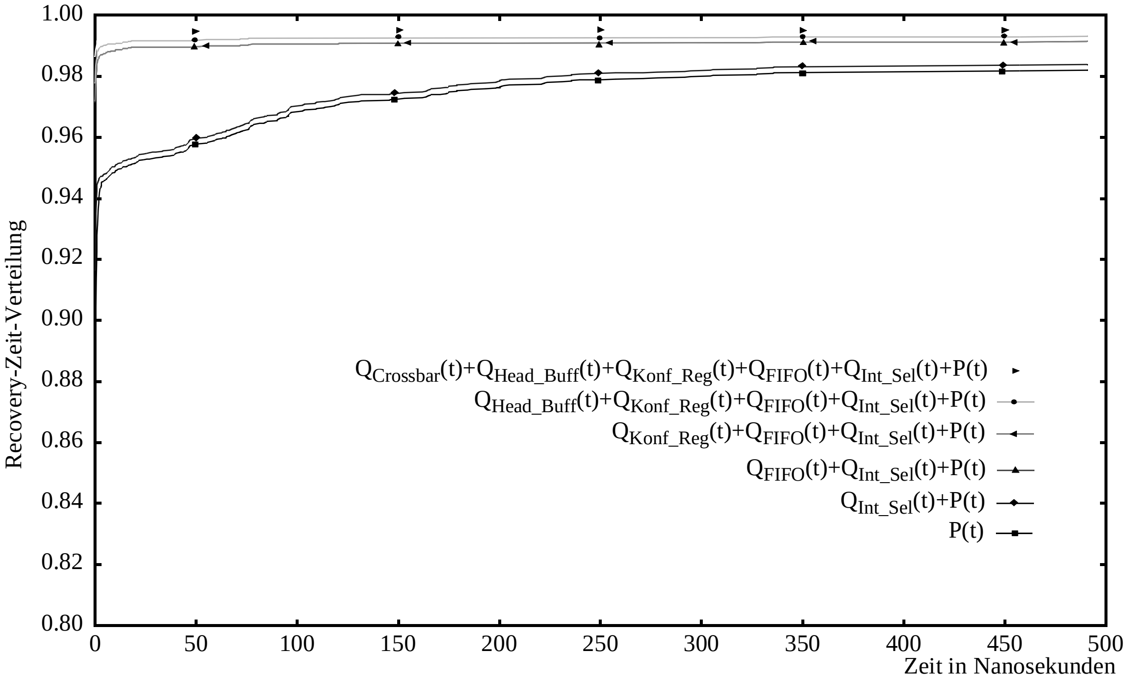


Abb. 4-12: Recovery-Zeiten noch zu behebender Übersprechfehler des STC104 (0-500ns)

In der folgenden Abb. 4-13 ist für das Fehlermodell der Übersprechfehler die Abhängigkeit der Crash-Fehler und der behobenen Fehler von der Anzahl der möglichen Fehler im System veranschaulicht.

Wie beim Stuck-At Fehlermodell ist auch hier bei den Crash-Fehlern (linker Balken) zu erkennen, daß der *Interval Selector* trotz seines hohen Anteils an möglichen Fehlern nur wenig zu den Crash-Fehlern des Systems beiträgt. Bemerkenswert ist hierbei die in Abb. 4-11 und Abb. 4-12 schon zu erkennende Tatsache, daß die Fehler in den *Konfigurationsregistern* in so gut wie keinem Fall (*0.1%*) zu Crashes des Systems führen. Dies ist eine Umkehrung des Verhaltens gegenüber dem Stuck-At Fehlermodell, bei dem eine überproportionale Abhängigkeit bei dieser Komponente zu erkennen war. Bei genauerer Betrachtung der Komponente und deren Fehler, kann dieses Verhalten mit der Tatsache erklärt werden, daß es hierbei zwar zu einem Übersprechen von Signalleitungen kommen kann, die Übernahme eines falschen Registerwerts jedoch nur durch eine Änderung des Clock-Eingangswertes von ‘0’ nach ‘1’ erfolgen kann. Dies ist jedoch im Vergleich zu allen möglichen Fehlerkombinationen innherhalb der Komponente ein nur sehr seltener Fall. Betrachtet man den Anteil der FSMs an den Crash-Fehlern, so erkennt man analog zum Stuck-At Fehlermodell eine überproportionale Abhängigkeit. Wie beim Stuck-At Fehlermodell manifestiert sich der Fehler in den Zustandsregistern und führt dort zu einem nicht reparablen Fehlverhalten des Systems.

Vergleicht man die Anzahl möglicher Fehler bei den verschiedenen Fehlermodellen (jeweils mittlere Balken von Abb. 4-8 und Abb. 4-13), so erkennt man eine weitgehende Übereinstimmung der Anteile der einzelnen Komponenten. Einzig die *Crossbar*-Komponente hat mit *12.6%* bei den Übersprechfehlern eine hervorgehobenere Stellung als beim Stuck-At Fehlermodell (*7.5%*).

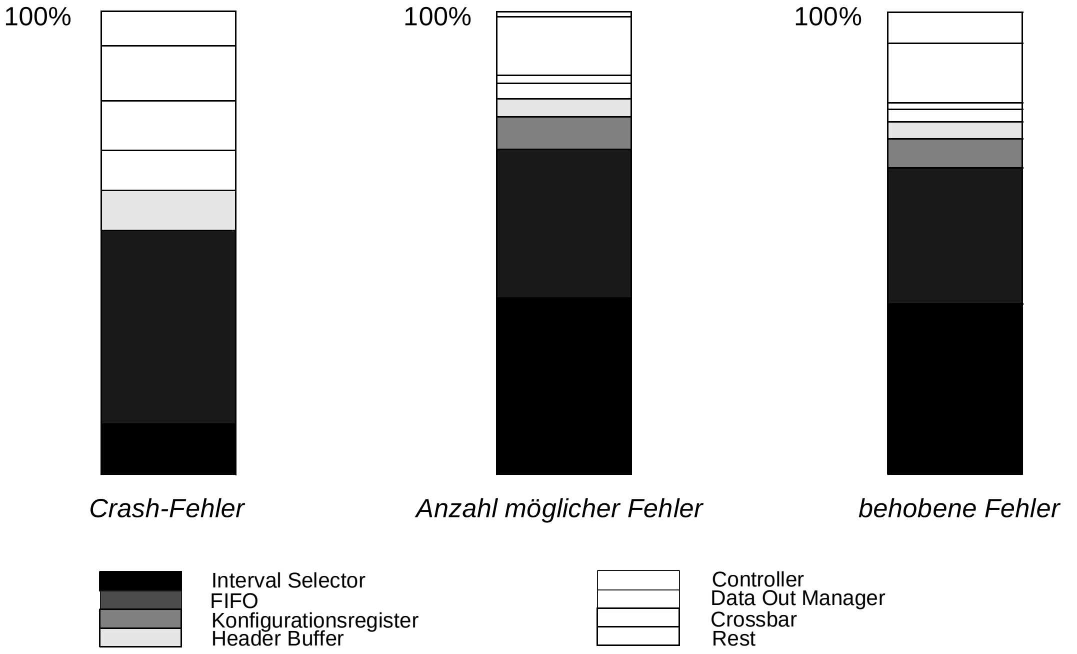


Abb. 4-13: Vergleich der Fehler-, Crash- und Recoverywahrscheinlichkeiten des STC104 bei Übersprechfehlern

Der Anteil behobener Fehler der jeweiligen Komponenten des STC104 entspricht wieder, wie beim Stuck-At Fehlermodell, in etwa dem Anteil möglicher Fehler für diese Komponente. Einzig für die mit *Rest* bezeichnete Zusammenfassung kleinerer Komponenten zeigt sich, wie auch bei den Crash-Fehlern, eine bedeutendere Rolle als es der Anteil möglicher Fehler erwarten läßt.

4.2.3 Vergleich der Fehlermodelle

Die schon beim Stuck-At Fehlermodell beobachtete Tatsache, daß vom jeweiligen Anteil möglicher Fehler nicht direkt auf das Crash-Verhalten der Komponenten zurückzuschließen ist, soll mit der folgenden Abb. 4-14 genauer untersucht werden. In Kapitel 4.2.1 wurde die Vermutung geäußert, daß Crashes eher mit dem Anteil an Registern, d.h. Gattern in denen sich ein Fehler für längere Zeit manifestieren kann, zusammenhängt. In Abb. 4-14 ist zu diesem Zweck im mittleren Balken der Anteil der einzelnen Komponenten am Gesamtregisteraufkommen angegeben. Erwartungsgemäß ist der größte Teil der Register in der Komponente der *Konfigurationsregister* zu finden. Darauf folgt die *FIFO* mit ihren 20 jeweils 10 Bit breiten Registern, zu denen noch die Register der *FIFO*-Steuerung hinzukommen. Am anderen Ende der Skala befindet sich der *Interval Selector*, dessen Hauptaufgabe in der Berechnung des Intervalsegments für den jeweils aktuellen Header besteht.

Betrachtet man nun die Crashfehler des STC104, so kann man für den *Interval Selector* und für die *FIFO* einen annähernd gleichen Anteil (bezogen auf den jeweiligen Registeranteil) erkennen. Obwohl erwartungsgemäß die meisten Crashfehler des Systems von den *Konfigura-*

tionsregistern ausgehen, ist hier der Anteil jedoch geringer als der zugehörige Registeranteil. Ein gegenläufiger Trend kann beim Stuck-At Fehlermodell für die FSM-basierten Komponenten beobachtet werden. Hier ist der Anteil an Crashfehlern jeweils deutlich größer als es der Registeranteil erwarten läßt.

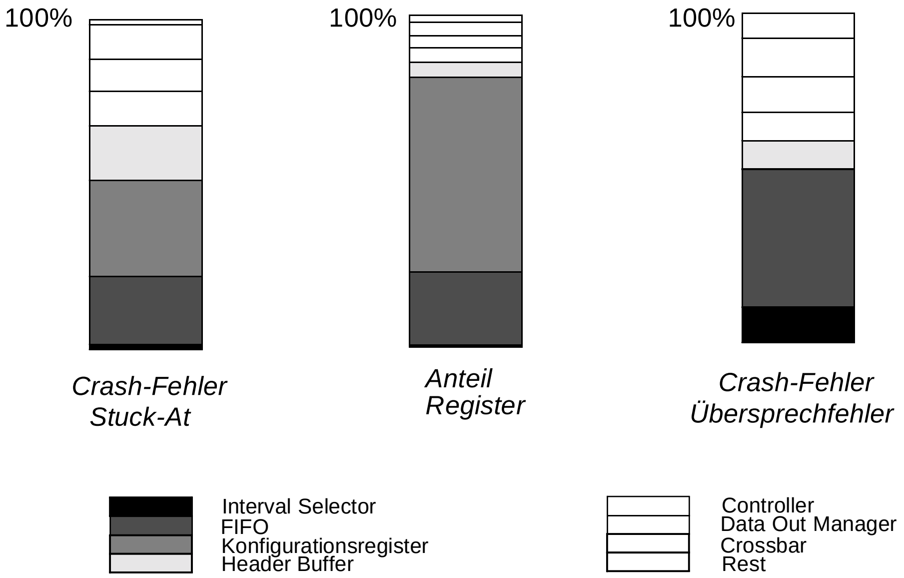


Abb. 4-14: Zusammenhang von Crashfehlern und Komponentenaufbau in Abhängigkeit von den Fehlermodellen

Besonders für die ersten drei Komponenten ergibt sich für das Übersprechfehlermodell jedoch ein deutlich anderes Bild als beim Stuck-At Fehlermodell. Hier spielen die Fehler im *Interval Selector* eine wesentlich größere Rolle als beim Anteil an den Gesamtregistern. Dasselbe gilt hierbei für Crash-Fehler in der *FIFO*, während es bei den *Konfigurationsregistern* aus den oben genannten Gründen trotz des hohen Registeranteils zu keinen nennenswerten Crashes gekommen ist.

Mit diesen Betrachtungen läßt sich aufzeigen, daß das Crash-Verhalten einer Komponente des Systems nur in sehr begrenztem Maße vom statischen Aufbau der Komponente abhängt. Im Einzelfall spielt also die Funktion und damit das Zusammenspiel der einzelnen Komponenten bzw. die Häufigkeit ihrer Verwendung eine weit entscheidendere Rolle als die Häufigkeit des Vorkommens einzelner Subkomponenten. Bemerkenswert ist hierbei, daß dies nicht nur für die Häufigkeit des Vorkommens der Register gilt, sondern auch für den Anteil der Komponente am Gesamtaufkommen der Fehlerquellen. Im Gegensatz dazu ist jedoch die Wahrscheinlichkeit, daß ein Fehler selbständig vom System behoben wird für beide Fehlermodelle ungefähr gleich dem Anteil am Gesamteinkommen der Fehlerquellen.

Die zeitliche Dynamik des Recovery-Verhaltens der Komponenten weicht bei den beiden Fehlermodelle nur unwesentlich voneinander ab. Insgesamt ist die Wahrscheinlichkeit eines Systemcrashes jedoch bei einem Stuck-At-Fehler größer als bei einem Übersprechfehler. Die Aufteilung dieser Crash-Fehler auf die einzelnen Komponenten weicht für beide Fehlermodelle erheblich voneinander ab.

In der folgenden Abb. 4-15 wird für beide Fehlermodelle die Wahrscheinlichkeit für einen Crash-Fehler in Abhängigkeit von der Dauer der Fehlerinjektion (d.h. der Länge der Phase Ph_1) aufgezeigt. Hierbei ist vor allem während der ersten *10 ns* ein stark unregelmäßiger Verlauf der Kurven zu erkennen. Obwohl man vermuten könnte, daß mit einer Zunahme der Fehlerinjektionsdauer in jedem Fall eine Erhöhung der Wahrscheinlichkeit eines Crash-Fehlers einhergeht, ist bei Fehlerdauern von weniger als *10 ns* öfters sogar ein entgegengesetzter Trend zu verzeichnen. Da die Fehlerdauer bei VERIFY exponentiell verteilt ist und es damit mehr Fehlerinjektionen mit kurzen Fehlerdauern gibt, ist dieser Trend auch nicht mit Meßfehlern auf Grund einer zu geringen Stichprobe in diesem Bereich zu erklären.

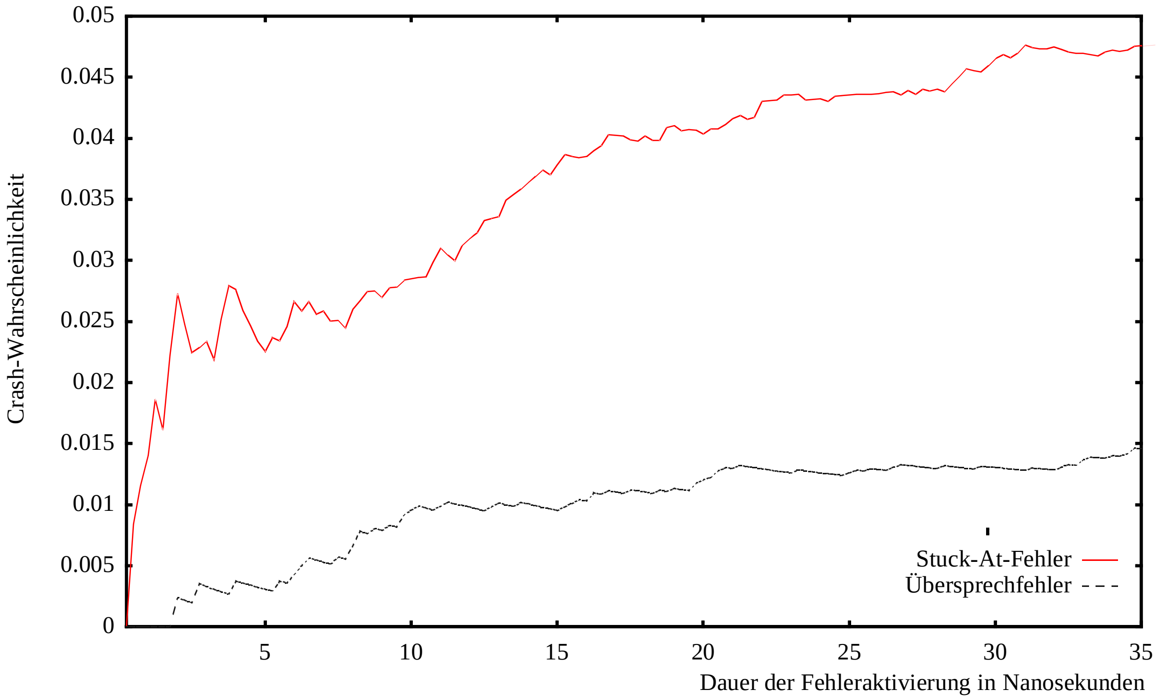


Abb. 4-15: Wahrscheinlichkeit eines Crashfehlers beim STC104 nach Injektionsdauer (bis 35ns)

Eine mögliche Erklärung für dieses Verhalten könnte im mittleren zeitlichen Abstand eines injizierten Fehlers zu dem nächstgelegenen Register sein. Sobald eine Fehlerdauer lang genug ist, damit sich der Fehler vom injizierten Gatter ausgehend in einem Register manifestieren kann, führt eine weitere Verlängerung der Fehlerinjektionsdauer zu keiner Erhöhung der Crash-Wahrscheinlichkeit mehr. Erst wenn das fehlerhafte Signal während seiner Fehleraktivierungszeit das übernächste Register erreicht, erhöht sich wieder sprunghaft die Wahrscheinlichkeit eines Crashfehlers.

Wie schon bei den vorhergehenden Untersuchungen gezeigt wurde, ist beim Übersprechfehlermodell die Gesamtwahrscheinlichkeit eines Crashfehlers geringer als beim Stuck-At-Fehlermodell. Dies ist auch die Erklärung für die Beobachtung aus Abb. 4-15, daß die Kurve für die Wahrscheinlichkeit eines Crashfehlers in Abhängigkeit von der Fehlerinjektionsdauer bei Übersprechfehlern geringer ist als bei Stuck-At-Fehlern.

In Abb. 4-16 ist der weitere Verlauf der Kurven aus Abb. 4-15 aufgezeichnet. Es ist zu erkennen, daß eine Verlängerung der Fehlerinjektionsdauer über *100 ns* bei beiden Fehlermodellen zu keiner Erhöhung der Crash-Wahrscheinlichkeit mehr führt.

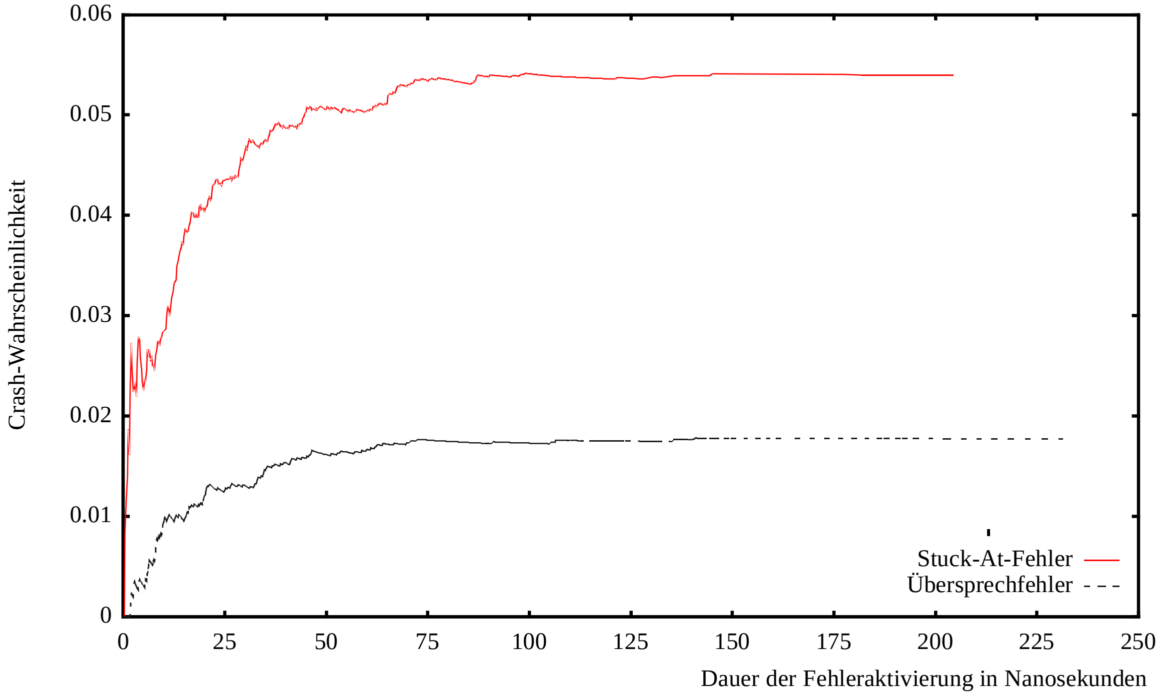


Abb. 4-16: Wahrscheinlichkeit eines Crashfehlers beim STC104 nach Injektionsdauer (bis 250ns)

4.3 Recovery-Verhalten des ATM-Switches

Im Gegensatz zum STC104 wurden im Modell der Knockout-Architektur des ATM-Switches keine Übersprechfehler injiziert. Desweiteren beschränkt sich die Aufgliederung der Meßergebnisse auf nur drei Subkomponenten des Switches. Wie man in den folgenden Abbildungen erkennen kann, ist dies durch die Dominanz der *Shared Buffer* Komponente begründet.

Für die Untersuchungen des Recovery-Verhaltens des ATM-Switches wurde die gleiche Gatterbibliothek wie bei den Untersuchungen des STC104 und damit auch die gleichen Fehlertypen (Stuck-At-0 bzw. Stuck-At-1 an jeder Ein- bzw. Ausgangsleitung) gewählt. Die exponentiell verteilte Fehlerdauer hat hierbei wieder einen Mittelwert von *25 ns* und es wurden wie beim STC104 auch hier *5000* Fehlerinjektionen durchgeführt. Bedingt durch die verschiedene Struktur wurde eine Fehlerbeobachtungszeit von *2500 ns* gewählt.

4.3.1 Stuck-At Fehler

In Abb. 4-17 und Abb. 4-18 sind die Recovery-Zeiten bereits behobener Stuck-At Fehler beim Knockout-Switch aufgetragen. Wie beim STC104 wurde auch hier der Zeitabschnitt der ersten *5* Nanosekunden besonders betrachtet. Wie oben schon erwähnt wurde, ist hierbei der *Shared Buffer* die dominierende Komponente der automatischen Fehlerbehebung. Mit *57.7%* hat diese Komponente den Hauptanteil an Fehlern, die direkt nach Beendigung der Fehlerinjektion keinerlei Auswirkungen auf das System haben. Mit nur jeweils *0.7%* und *2.4%* folgen hierbei der *Concentrator* und die restlichen Komponenten des Knockout-Switches. Schon nach *0.5 ns* (d.h. *8* Gatterlaufzeiten) ist für diesen Zeitraum so gut wie keine Dynamik mehr zu erkennen. Während der ersten *0.5 ns* sind es vor allem Fehler innerhalb des *Shared Buffer*, die vom System automatisch behoben werden können. Der Anteil behobener Fehler in dieser Komponente verbessert sich bis dahin um *13.6%* auf *71.3%*. Bis zum Ende des betrachteten Zeitraums von *5ns* verbessert sich dieser Wert nur noch um *0.4%*.

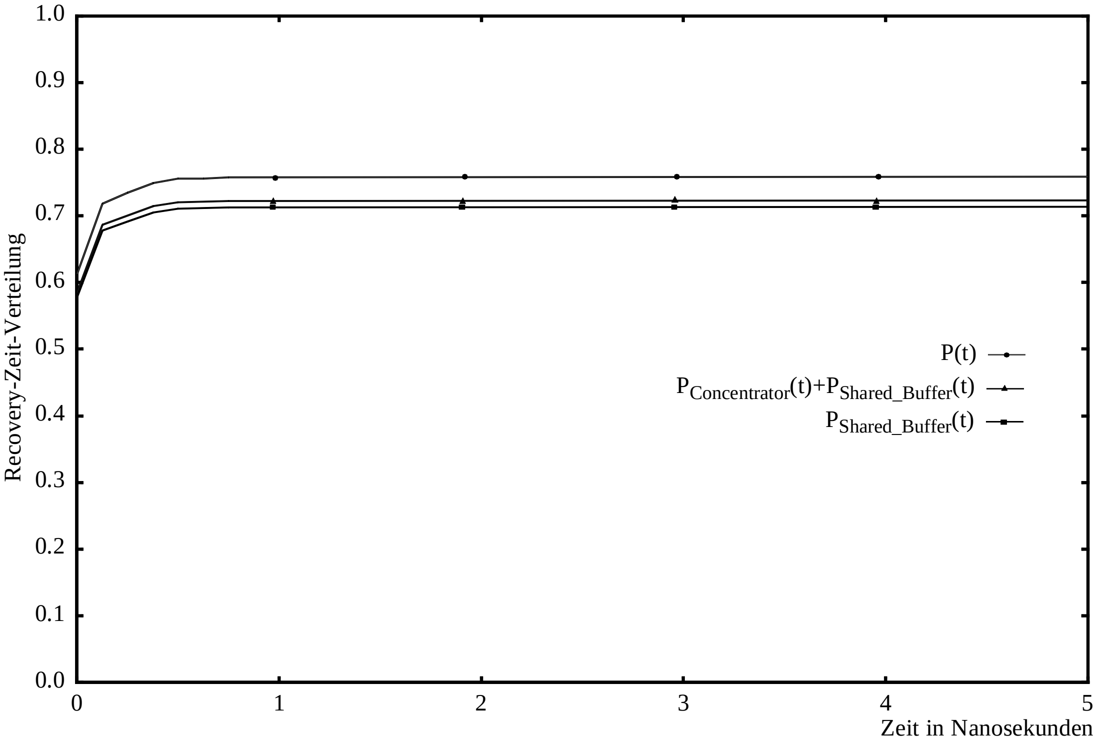


Abb. 4-17: Recovery-Zeiten bereits behobener Stuck-At-Fehler des ATM-Switches (0-5ns)

Der weitere Verlauf der Kurve bis hin zu *2500 ns* wird in Abb. 4-18 dargestellt. Im Gegensatz zum Stuck-At Fehlermodell des STC104 erkennt man hier eine starke Dynamik des Recovery-Verhaltens beim Knockout-Switch über einen längeren Zeitraum. Erst nach *2000 ns* flacht die Kurve ab und endet bei einer Gesamtwahrscheinlichkeit für ein Recovery des Systems von *94.4%*. Die Dynamik innerhalb der ersten *2000 ns* ergibt sich vor allem aus dem Verhalten des *Shared Buffers*, der damit eine insgesamte Dynamik von *57.7%* bis *86.7%* aufweist. Diese Dynamik ist durch den Aufbau und die Semantik dieser Komponente begründet. Die Hauptaufgabe des *Shared Buffers* besteht in der Zwischenspeicherung der ATM-Zellen in mehreren

FIFO-Puffern. Ein dort eingebrachter Fehler trifft also mit großer Wahrscheinlichkeit ein Gatter, das im Datenpfad einer ATM-Zelle liegt. Dies ist damit auch die Begründung für die kontinuierliche, fast lineare Verbesserung des Recovery-Verhaltens dieser Komponente, da Fehler, die ausschließlich die zu transportierenden Daten betreffen im Zeitraum bis spätestens *2000 ns* aus dem Switch herausgeleitet werden, ohne dort weitere Schäden zu hinterlassen.

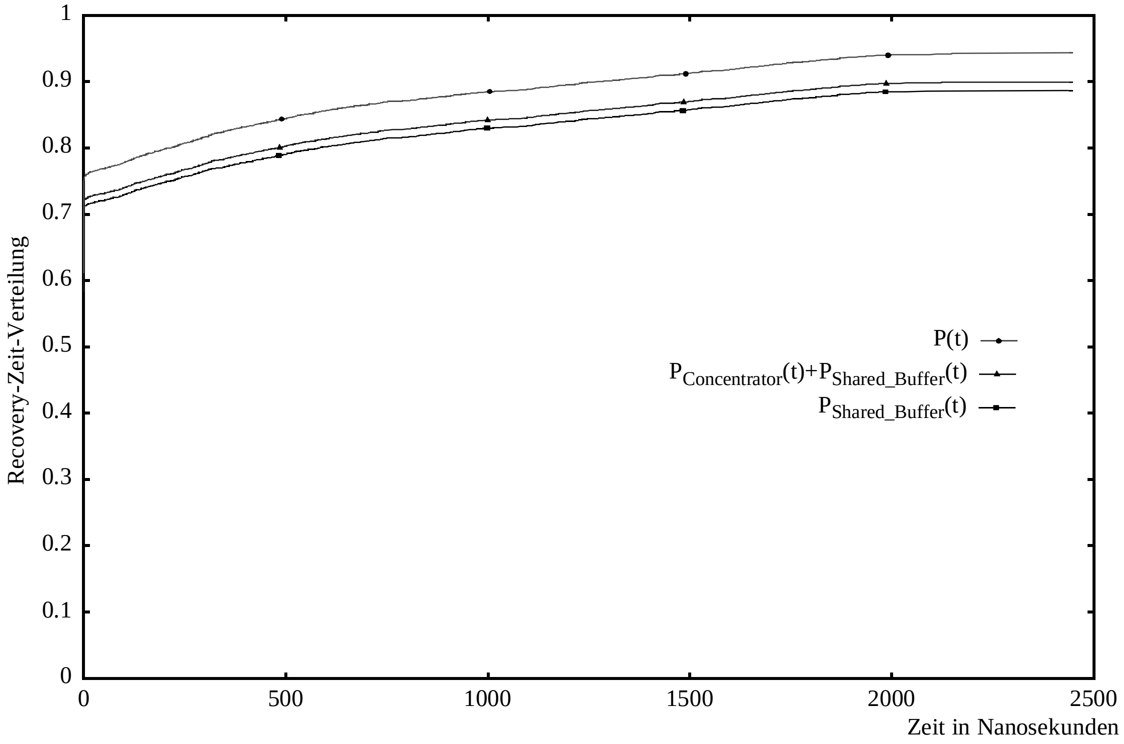


Abb. 4-18: Recovery-Zeiten bereits behobener Stuck-At-Fehler des ATM-Switches (0-2500ns)

Das Recovery-Verhalten noch zu behebender Fehler des Knockout-Switches im Zeitraum von *0 ns* bis hin zu *5 ns* ist in Abb. 4-19 dargestellt. Wie in den vorangegangenen Abbildungen schon zu erkennen war, werden in den ersten *0.5 ns* vor allem Fehler des *Shared Buffer* behoben. Insgesamt ist der Anteil noch zu behebender Fehler für diese Komponente am größten. Auch hier ist für den Zeitraum *0.5 ns* bis *5 ns* keine erwähnenswerte Dynamik zu erkennen.

In Abb. 4-20 ist der weitere Verlauf der Kurve bis hin zu *2500 ns* dargestellt. Auch hier ist wieder zu erkennen, daß sich der Anteil noch zu behebender Fehler hauptsächlich in der Komponente des *Shared Buffer* verbessert. Die restlichen Komponenten tragen so gut wie gar nicht zum Recovery des Systems bei. Bemerkenswert ist auch in dieser Abbildung das Einpendeln des Anteils noch zu behebender Fehler nach *2000 ns*.

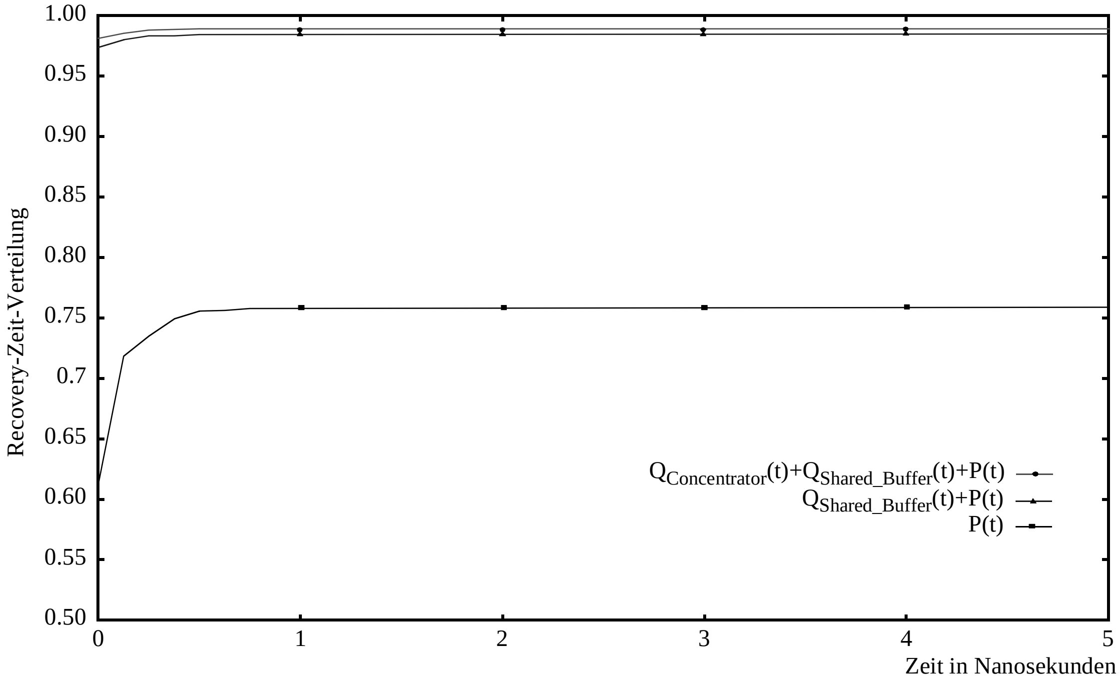


Abb. 4-19: Recovery-Zeiten noch zu behebender Stuck-At-Fehler des ATM-Switches (0-5ns)

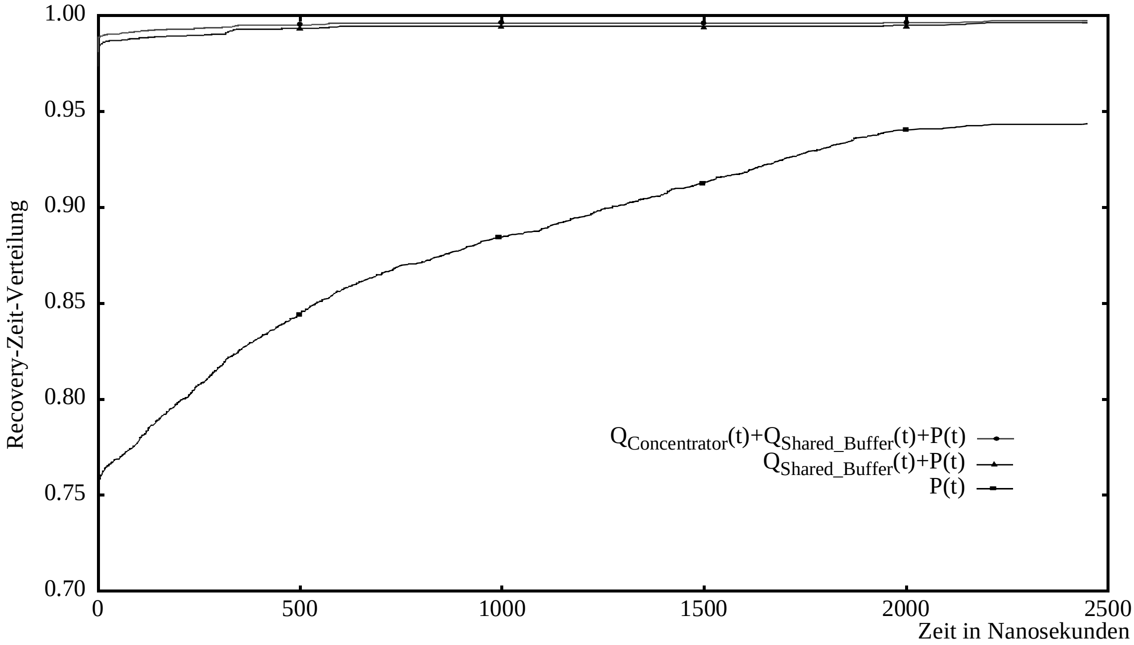


Abb. 4-20: Recovery-Zeiten noch zu behebender Stuck-At-Fehler des ATM-Switches (0-2500ns)

Eine genauere Betrachtung der Abhängigkeit der Crash- bzw. der Recovery-Wahrscheinlichkeiten der einzelnen Komponenten des Knockout-Switches vom Anteil am Gesamtaufkommen aller möglichen Fehler ist in Abb. 4-21 gegeben. Wie schon bei den Untersuchungen des STC104 ist hier im linken Balken für jede Komponente die Wahrscheinlichkeit angegeben, daß ein Crash-Fehler des Systems durch eine Fehlerinjektion innerhalb dieser Komponente begründet ist.

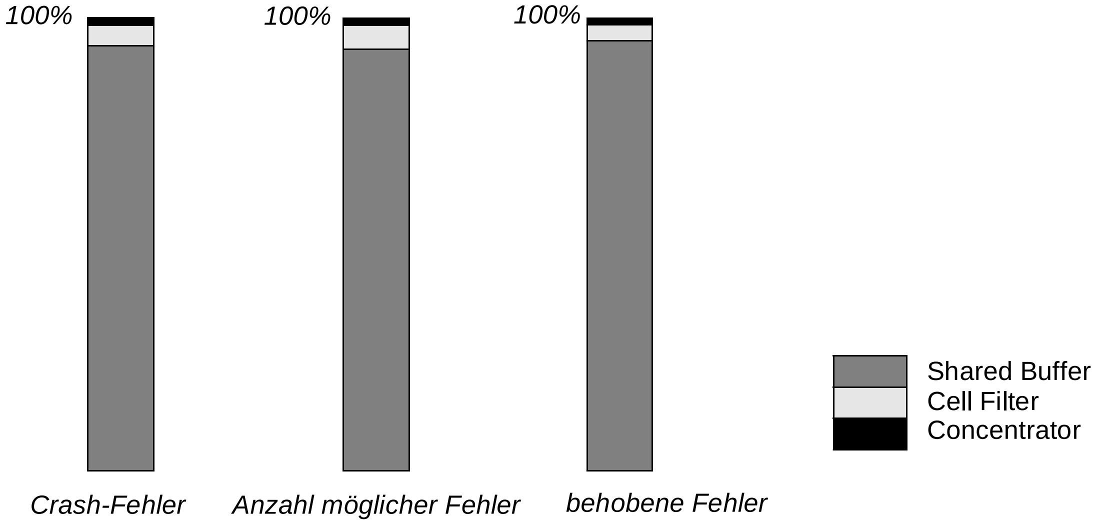


Abb. 4-21: Vergleich der Fehler-, Crash- und Recoverywahrscheinlichkeiten des ATM-Switches bei Stuck-At-Fehlern

Hierbei ist deutlich zu erkennen, daß dieser Anteil deutlich mit dem Anteil am Gesamtaufkommen möglicher Fehler korreliert ist. Im rechten Balken der Abb. 4-21 ist für jede Komponente der Anteil behobener Fehler dargestellt. Auch hier ist eine deutliche Korrelierung mit dem Anteil an den Fehlerquellen zu erkennen. Insgesamt verhält sich die Komponente des *Shared Buffer* somit ähnlich der *FIFO* des STC104, was vor allem durch den ähnlichen Aufbau und der ähnlichen Semantik der Komponente begründet ist.

Analog zu den Untersuchungen beim STC104 wurde auch hier die Abhängigkeit der Wahrscheinlichkeit für einen Crashfehler von der Dauer der Fehlerinjektion (Länge der Phase Ph_1) ausgewertet (Abb. 4-22). Wie beim STC104 ist auch hier ein stark schwankendes Verhalten der Kurve zu beobachten. Bei der Knockout-Architektur des ATM-Switches ist das Überschwingverhalten dieser Abhängigkeit bei Fehlerdauern von bis zu *20 ns* jedoch weit ausgeprägter als beim STC104. Die Amplitude dieser Schwingung wird auch hier geringer, wobei sie sich im Gegensatz zu dem Verhalten beim STC104 auf einen deutlich unterhalb der Maximalwahrscheinlichkeit liegenden Wert einpendelt.

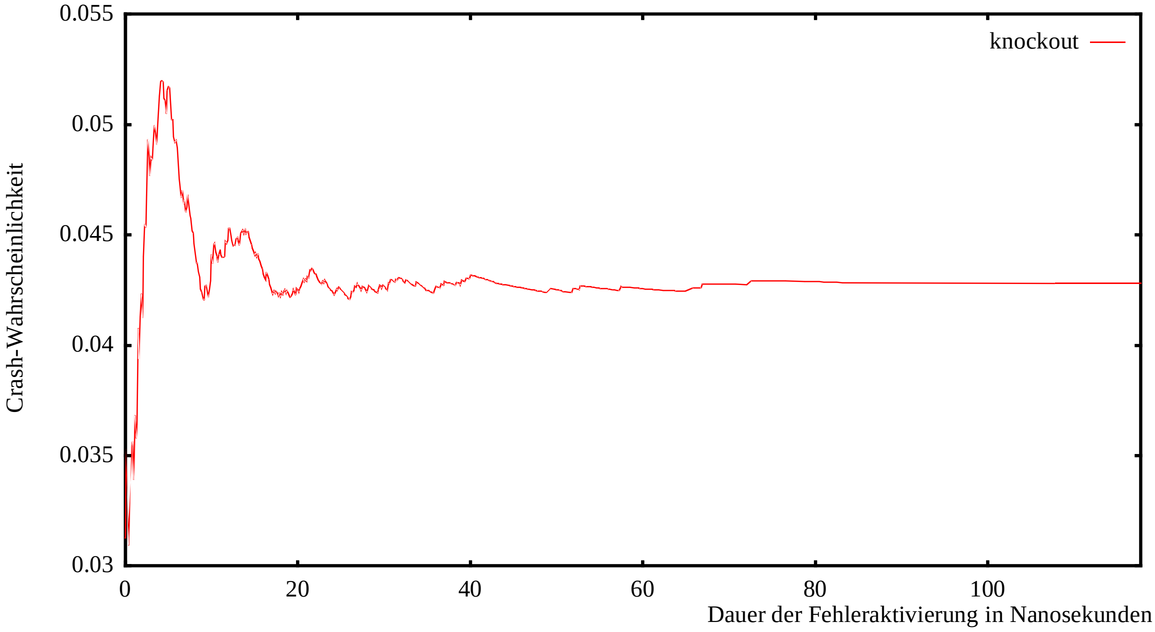


Abb. 4-22: Wahrscheinlichkeit eines Crashfehlers beim ATM-Switches nach Injektionsdauer

Zum Abschluß dieses Kapitels soll jetzt noch eine Kurve präsentiert werden, die die Verteilung der Crashfehler in Abhängigkeit von der Fehlerinjektionsdauer angibt. Hierbei wurde das Verhalten aller drei Systeme (STC104 mit den beiden Fehlermodellen und der ATM-Switch mit dem Stuck-At-Fehlermodell) in einem Diagramm zusammengefaßt.

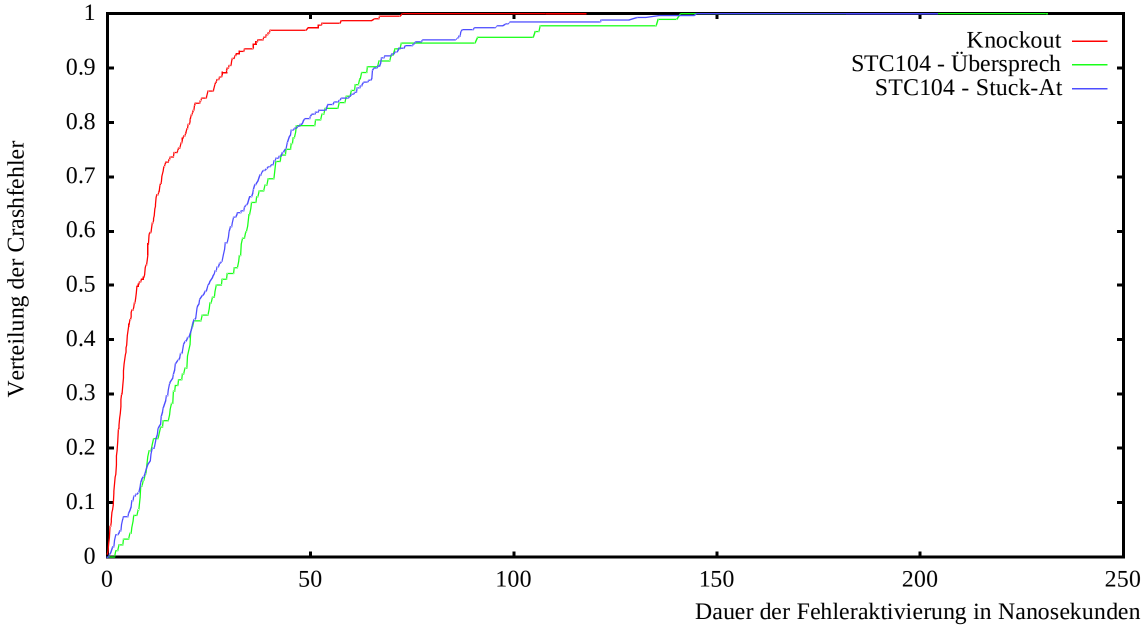


Abb. 4-23: Verteilung der Crashfehler nach Injektionsdauer

An den Kurvenverläufen ist zu erkennen, daß die unterschiedlichen Fehlermodelle beim STC104 keinen Einfluß auf die Verteilung der Crashfehler haben. Insgesamt ist jedoch beim ATM-Switch ein höherer Prozentsatz eingetretener Crashfehler bei gleicher maximaler Fehlerinjektionsdauer dargestellt. Die Erklärung dieses Verhaltens liegt im fast doppelt so hohen Takt des ATM-Systems. Damit ist bei der gleichen Fehlerinjektionsdauer beim ATM-Switch schon viel früher ein Register erreicht, als beim STC104.

Insgesamt läßt sich aus dem Diagramm ableiten, daß beim ATM-Switch über *90%* aller Fehler, die zu einem Crash geführt haben, eine Fehlerinjektionsdauer von weniger als *30 ns* haben. Beim STC104 ist diese *90%* Marke erst bei einer Fehlerinjektionsdauern von *65 ns* erreicht.

KAPITEL
5

Protokollverhalten im Fehlerfall

Im Gegensatz zu den Untersuchungen des vorhergehenden Kapitels soll in den folgenden Abschnitten nicht mehr das Fehlerverhalten auf Gatterebene betrachtet werden, sondern ausschließlich der Einfluß der Fehler auf die Funktionalität der betrachteten Systeme. Aus diesem Grund wird im Bezug auf die beiden realisierten Protokolle untersucht, in wieweit der Protokollablauf beeinträchtigt wird, und in welchem Maß die zu transportierenden Daten von den Fehlern betroffen sind.

Dazu wurde neben den bisher betrachteten temporären Fehlern, d.h. einmalig auftretenden Fehlern mit einer begrenzten, über das Fehlermodell vorgegebenen Fehlerdauer, auch permanente Fehler untersucht. Bei dem in Kapitel 4.1 auf Seite 45 angegebenen Modell der Phasen eines Fehlerinjektionsexperiments fallen dabei Ph_2 und Ph_3 weg, da es keine Recovery bzw. keinen fehlerfreien Lauf nach Aktivierung des Fehlers gibt. Die Phase Ph_1 der Fehleraktivierung ist so lange gewählt, daß jeweils mehrere eingehende Protokolleinheiten, d.h. Pakete beim STC104 bzw. Zellen beim ATM-Switch zum Ausgangslink weitergeleitet werden. Damit ist sichergestellt, daß ein Fehler an einer beliebigen Stelle des Datenpfades innerhalb des Switches bzw. an einer den Datenpfad kontrollierenden Einheit am Ausgangslink sichtbar wird.

5.1 Protokollverhalten bei Fehlern im STC104

In einem ersten Schritt soll die insgesamte Fehlerhäufigkeit auf Protokoll- bzw. Datenebene bezogen auf die beiden Hauptkomponenten des STC104 dargestellt werden. Dazu wurden jeweils *5000* temporäre und *5000* permanente Fehler in einem *Link* des STC104 injiziert und anschließend genauso mit dem *Crossbar* verfahren[1]. Desweiteren wurden in diese beiden Komponenten jeweils *5000* temporäre und *5000* permanente Übersprechfehler injiziert. Es sei in diesem Zusammenhang noch einmal darauf hingewiesen, daß bei jeder der *5000* Fehlerinjek-

---

[1] Für den detaillierten Aufbau des STC104 sei auf Kapitel 3.1.2 auf Seite 24 verwiesen.

tionen pro Experiment immer nur ein einziger Fehler injiziert wird. Nach jeder Fehlerinjektion läuft das System für einen vorgegebenen Beobachtungszeitraum weiter, um die Fehlerauswirkungen zu untersuchen. Für diesen Beobachtungszeitraum wurde bei temporären Fehlern wie bei den in Kapitel 4 durchgeführten Untersuchungen auf der Gatterebene ein Wert von *500 ns* und bei permanenten Fehlern ein Wert von *1000 ns* gewählt.

Wie bei der Vorstellung des Lastmodells in Kapitel 3.3.1 auf Seite 39 schon erwähnt wurde, hängt auch für diese Untersuchung an der Netzwerkseite jedes *Links* des STC104 ein Lastgenerator. Jeder Lastgenerator wird dabei nicht nur zum Generieren und Senden der Pakete, Token und Bits verwendet, sondern dient auch gleichzeitig als Empfangsmodul der Daten, die von dem ihm zugeordneten Link des STC104 ins Netz weitergegeben werden. Beim Empfang wandelt der jeweilige Lastgenerator die Bits in Token um und legt die empfangenen Daten als interpretierte Token zusammen mit einem Zeitstempel in einer Datei ab. Wie bei der Vorstellung von VERIFY in Kapitel 2.2.2 dargelegt wurde, besteht ein Experiment mit *N* zu injizierenden Fehlern immer auch aus dem Golden Run, d.h. dem fehlerfreien Lauf über den gesamten Beobachtungszeitraum. Während dieses Golden Run speichert jeder Lastgenerator, genauso wie im Fehlerfall, seine empfangenen Daten in einer Datei. Während des Beobachtungszeitraums jeder Fehlerinjektion werden die empfangenen Daten zusammen mit einem eindeutigen Kennzeichner für diese Injektion ebenfalls abgespeichert. Damit können die Daten des Golden Run als Referenz herangezogen werden, um die vom STC104 im Fehlerfall weitergeleiteten Daten zu analysieren[1].

5.1.1 Wahrscheinlichkeit eines Protokollfehlers

Der Anteil derjenigen Experimente, bei denen nach einer Fehlerinjektion ein Fehler im Protokoll oder in den Daten der Pakete bzw. Zellen zu finden war, ist in Abb. 5-1 dargestellt.

Im Diagramm ist zu erkennen, daß permanente Fehler, wie zu erwarten, eine höhere Wahrscheinlichkeit eines Protokoll- bzw. Datenfehlers beherbergen als temporäre Fehler. Trotz allem ist es jedoch erstaunlich, daß selbst bei permanenten Fehlern nur zwischen *5%* und *35%* aller injizierten Fehler eine Auswirkung auf die Protokoll- bzw. Datenebene haben. Dies deutet auf eine hohe Latenzzeit, d.h. die Zeit zwischen dem physikalischen Auftreten des Fehlers und der Störung der Funktionalität des Systems hin. Daraus ist zu schließen, daß die Latenzzeit schon auf dieser Ebene eine nicht zu unterschätzende Größe erreichen kann[2]. Diese nach außen (noch) nicht sichtbaren Fehler können sich dadurch im inneren des Systems verbreiten und im Moment der äußeren Manifestation durch Fehler im Protokoll oder in den Daten zu erheblich größeren Schäden führen, als dies bei einer rechtzeitigen Erkennung und Eingrenzung des Fehlers möglich wäre.

---

1. Die Analyse wurde von einem ebenfalls in dieser Arbeit entwickelten C-Programm nach dem Ende aller Fehlerinjektionen eines Experimente durchgeführt.
2. Betrachtet man ein komplettes Netzwerk aus vielen Switches, so können fehlerhafte Daten durch mehrere Dutzent STC104 geleitet werden, bis ein Fehler überhaupt erkannt werden kann. Damit ist bei dieser gesamtheitlichen Betrachtung eine noch viel größere Latenzzeit zu erwarten, da Datenfehler bedingt durch das Protokoll des STC104 erst beim Zielprozessor erkannt und behandelt werden können.

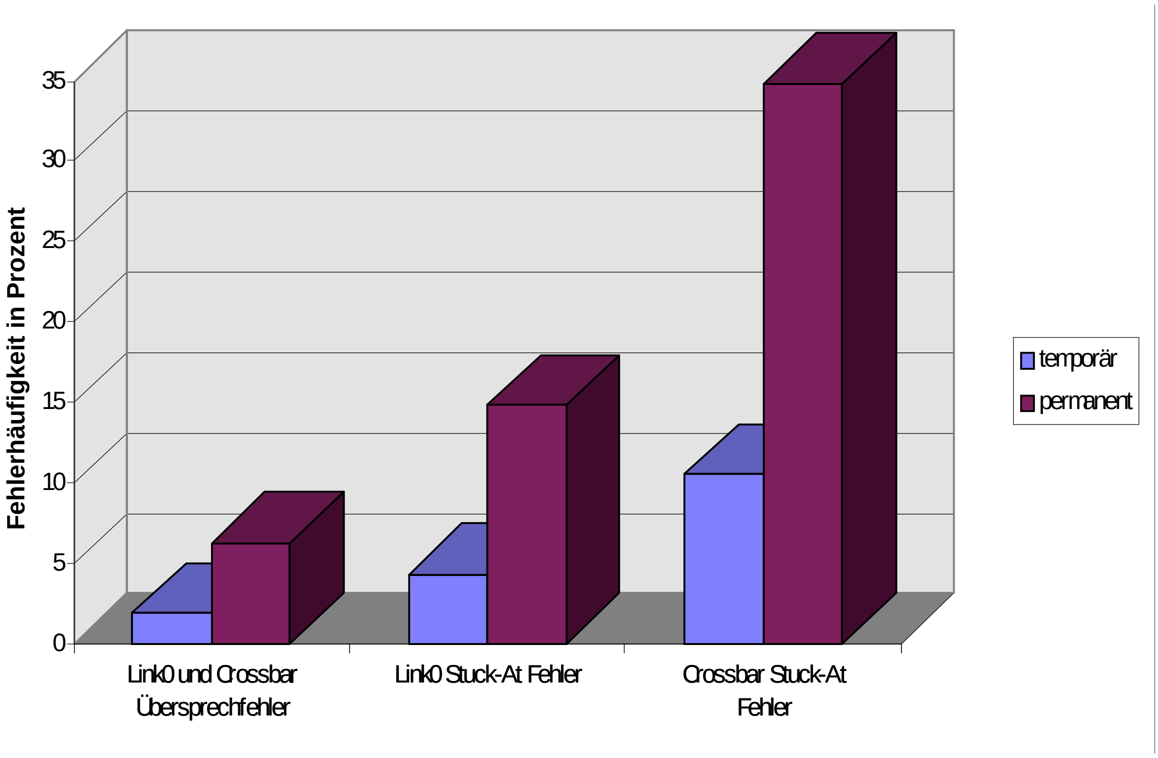


Abb. 5-1: Wahrscheinlichkeit eines Protokollfehlers nach Fehlerinjektion

Weiterhin ist bei den permanenten Fehlern zu beobachten, daß die verbindende Komponente des *Crossbar* bei gleichem Fehlermodell eine über doppelt so große Wahrscheinlichkeit eines Protokoll- bzw Datenfehlers aufweist, wie das beim *Link* der Fall ist. Insgesamt ist hier auch noch bemerkenswert, daß das Fehlermodell der Übersprechfehler nochmals eine um den Faktor zwei geringere Fehlerwahrscheinlichkeit aufweist. Könnte man dies bei temporären Fehlern noch mit den Meßergebnissen der insgesamt geringeren Fehlerwahrscheinlichkeit begründen (siehe Kapitel 4.2.2 auf Seite 54), so ist dies bei den permanenten Fehlern nicht mehr möglich, da die Selbstheilungskraft des Systems keinen Einfluß haben sollte. Eine Erklärung für dieses Verhalten wäre jedoch möglicherweise darin zu finden, daß es speziell beim *Link* sehr viele Orte für Übersprechfehler gibt, die keine Auswirkung auf den aktuellen Datenfluß haben. Beispiele solcher Komponenten wären die nicht benutzten Teile des *Interval Selectors*, der ausschließlich während der Auswertung der Paketheader zur Ansteuerung des *Crossbar* verwendet wird. Ist der *Crossbar* für ein Paket einmal geschaltet, so wird der *Interval Selector* bis zum Eintreffen des nächsten Pakets nicht mehr benötigt und dessen ausgehenden Signale ausmaskiert. Desweiteren werden innerhalb des *Interval Selectors* alle Signale derjenigen Subkomponenten[1] ausmaskiert, die sich bei der Intervall-Auswertung eines eintreffenden Paketheaders für nicht zuständig erklären (siehe Kapitel 3.1.2 auf Seite 24).

Die Wahrscheinlichkeit nach einem temporären Fehler auf Signalebene einen Protokoll- oder Datenfehler an mindestens einem der Ausgänge des STC104 zu erhalten, ist um den Faktor

---

[1] Hierbei handelt es sich hauptsächlich um die *Comparatoren*.

drei geringer als bei permanenten Fehlern. Mit einer Häufigkeit von *10%* aller injizierten Fehler ist auch hier der *Crossbar* die Komponente mit der größten Wahrscheinlichkeit eines Protokoll- bzw. Datenfehlers. Insgesamt lassen sich jedoch dieselben qualitativen Aussagen bezüglich der Fehlermodelle und der Komponenten der Fehlerinjektion treffen, wie bei den permanenten Fehlern.

5.1.2 Anteil betroffener Ausgangslinks

In einem weiteren Schritt soll untersucht werden, wie sehr sich ein einzelner, ins System eingebrachter Fehler ausbreitet. Dazu ist in Abb. 5-2 die Wahrscheinlichkeit angegeben, mit der sich Protokoll- bzw. Datenfehler an einer gegebenen Anzahl von Ausgangslinks bemerkbar machen. In dem Diagramm wird wieder nach temporären und permanenten Fehlern unterschieden, sowie die Meßergebnisse nach Art des Fehlermodells und Fehlerinjektionsortes aufgeschlüsselt.

Dabei zeigt sich aus den Meßergebnissen, daß unabhängig von der Fehlerart bei zwischen *85%* und *95%* der Fehlerfälle nur ein einziger Ausgangslink betroffen ist. Eine auffällige Ausnahme bilden temporäre Stuck-At Fehler, die im *Crossbar* des STC104 injiziert wurden. Hierbei haben weniger als *70%* nur einen Ausganglink beeinflußt, während bei annähernd *30%* der beobachteten Protokoll- bzw. Datenfehler genau drei Ausgangslinks betroffen waren. Gleichzeitig wurde in diesem Fall weder an genau zwei noch an genau vier Ausgangslinks ein anormales Verhalten beobachtet. Desweiteren tritt diese Anomalie nur bei temporär injizierten Stuck-At Fehlern auf, während die Gesamtwahrscheinlichkeit mehr als einen Ausgangslink zu betreffen bei permanenten Stuck-At Fehlern im *Crossbar* nur *5%* beträgt. Bei Übersprechfehler ist dieses Verhalten weder bei temporären noch bei permanenten Fehlern beobachtbar.

Aus den Meßdaten der Abb. 5-2 läßt sich erkennen, daß das Fehlermodell (Stuck-At oder Übersprechfehler) keinen nennenswerten Einfluß auf die Anzahl betroffener Links hat. Desweiteren sinkt mit Ausnahme des oben beschriebenen Falls die Wahrscheinlichkeit mehrerer betroffener Ausgangslinks mit wachsender Anzahl der Links. Der leichte Anstieg bei vier betroffenen Links im Falle temporärer Stuck-At Fehler im *Link0* des STC104 dürfte in der zu geringen Häufigkeit der aufgetretenen Fälle begründet sein[1]. Zusammenfassend läßt sich sagen, daß bei zwischen *5%* und *30%* aller Fehlerinjektionsexperimente mit einem einzigen injizierten Fehler mehr als ein einziger Ausgangslink betroffen war. Damit breiten sich die Fehler in einem nicht unerheblichen Maß in einem Netzwerk aus. Hierbei sei darauf hingewiesen, daß die Paritybit-Sicherung auf der Bitebene des Protokolls bei diesen Fehlern nicht greifen konnte, da das Paritybit am Ausgangslink immer wieder neu generiert werden muß. Die Fehlererkennung beim Protokoll des STC104 läßt also keine Erkennung von Fehlern im inneren des STC104 zu, sondern schützt ausschließlich die Übertragungsstrecke zwischen den Netzelementen[2].

---

[1] Zweimal waren drei Ausgangslinks betroffen und fünfmal wurden in vier Ausgangslinks Fehler beobachtet.

[2] Zum Beispiel STC104 oder T9000

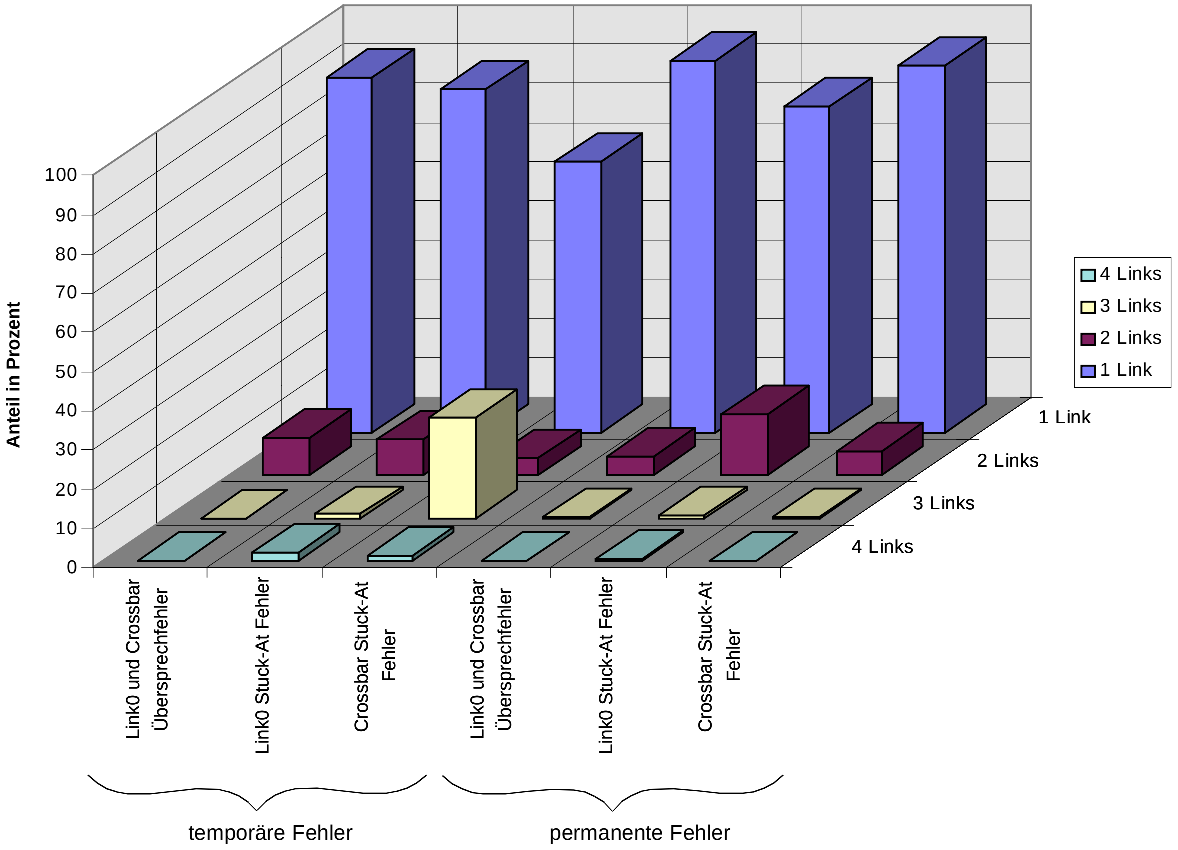


Abb. 5-2: Prozentsatz betroffener Links bei Protokollfehler

5.1.3 Protokollfehlertypen

Nach diesem Überblick sollen jetzt in einem weiteren Schritt die einzelnen Elemente des STC104-Protokolls genauer untersucht werden. Um die Auswirkungen der Protokollfehler beurteilen zu können, wird zuerst ein grober Überblick des Programmiermodells des T9000 gegeben[1]. Dieses legt fest, daß der Linkhardware bei jeder Kommunikation vom Empfangsprozeß ein Speicherbereich auf dem Empfangsprozessor zugewiesen wird, in dem die zu empfangenden Daten abgelegt werden können. Im Zusammenhang mit der *variable-length-communication* des T9000, wird aus den Header der empfangenen Daten der Empfangsprozeß bestimmt und die Daten in dem von ihm zugewiesenen Speicherbereich abgelegt. Dabei wird nicht auf die Größe der empfangenen Nachricht geachtet. Zur Kommunikation existiert ein Quittungsprotokoll auf End-zu-End Ebene, d.h. der Datenaustausch findet erst statt, wenn beide an der Kommunikation beteiligten Prozesse ihre Kommunikationsroutinen anstoßen. Ist der Empfangsprozeß zeitlich früher dran, wird die Information des Empfangswunsches zusammen mit dem oben erwähnten Speicherbereich der Linkhardware des Empfangsprozessors

---

[1] Detaillierte Information zum Programmiermodell des T9000 sind in [INM 93a] und [INM 93b] zu finden.

übergeben und der Empfangsprozeß blockiert sich, bis die Kommunikation vollständig stattgefunden hat. Ist der Sendeprozeß der erste, sendet dieser das erste Paket der Nachricht und die Linkhardware des Sendeprozessors wartet auf ein Acknowledge zum Senden weiterer Pakete bzw. zum Abschluß der Kommunikation, falls die Gesamtnachricht schon in diesem einen Paket Platz gefunden hat. Der Sendeprozeß wird daraufhin vom T9000 blockiert bis die gesamte Nachricht versendet worden ist. Der Empfangsprozessor hält für diesen Fall einen Puffer für genau ein Paket bereit. Ist der Empfangsprozeß dann bereit zum Empfangen der Nachricht, wird dieses erste Paket mit dem Senden einer ACK-Nachricht[1] zum Sendeprozessor quittiert. Der Sendeprozessor versendet dann das nächste Datenpaket und wartet wieder auf eine quittierende ACK-Nachricht. Diese wird vom Empfangsprozessor jedoch nicht erst nach dem Empfang des letzten Bytes eines Pakets versendet, sondern direkt nach Erhalt des ersten Bytes (d.h. des Headerbytes) einer Nachricht.

Zur genaueren Untersuchung der Fehlerauswirkungen ist in der Abb. 5-3 auf Seite 75 für die sechs schon bekannten Experimentreihen eine Aufschlüsselung der Fehler am Ausgangslink in folgende Kategorien erfolgt:

- **Keine Ausgabe**: ab einem gewissen Zeitpunkt wurden am Ausgangslink gar keine verwertbaren Protokolldaten mehr beobachtet. Alle Prozesse, die ihre Nachrichten über diesen Link verschicken, werden durch diesen Ausfalltyp blockiert. Da in einem größeren Netzwerk aus vielen STC104 Nachrichten sehr vieler Prozesse über einen gegebenen Link geroutet werden können, führt dieser Fehlertyp zu einer Blockade all dieser Prozesse, wobei diese Prozesse auf vielen Prozessoren verteilt sein können. Eine automatisierte Lokalisierung des Fehlers ist in diesem Fall sehr schwierig.

- **Datenfehler**: der Datenteil, d.h. Header- und/oder Datentoken des STC104-Protokolls waren verfälscht. Bei Headertoken heißt dies, daß die Nachricht zu einem falschen Empfänger weitergeleitet wird. Erwartet dieser ein Paket eines anderen Senders, so nimmt er die falsche Nachricht entgegen und sendet eine Quittungsnachricht an seinen eigentlichen Kommunikationspartner. Damit wird eine Kommunikation zwischen Prozessen gestört, deren Kommunikationspfad eventuell gar keine gemeinsamen Elemente mit demjenigen der fehlerhaften Kommunikation hat. Eine Lokalisierung der fehlerhaften Netzkomponente wird in diesem Fall extrem schwierig. Desweiteren fehlt dem Sender der Nachricht mit dem verfälschten Header ein ACK, womit sich diese beiden Prozesse blockieren.

- **Protokollfehler**: hierbei handelt es sich um Fehler bei den EOP- und/oder den EOM-Tokens. Auch diese Fehler können gravierende Auswirkungen für die Empfangsprozessoren bzw. die Empfangsprozesse haben. Wird ein EOP oder EOM-Token nicht oder erst verspätet gesendet, so werden alle bis dahin eingegangenen Datentoken vom Empfangsprozessor als Daten interpretiert, die zum aktuellen Paket gehören. Da das eigentliche Ende des aktuellen Pakets durch das fehlende EOP/EOM-Token nicht erkannt werden kann, wird das darauf folgende Datentoken nicht als Header interpretiert, wodurch die Kommunikation eines weiteren Prozeßpaares gestört wird. Desweiteren werden alle eingehenden

---

[1] Bestehend aus dem Headertoken dem direkt ein EOP-Token folgt

Datentoken in den, vom Empfangsprozeß zugewiesenen Speicher abgelegt. Da dieser im allgemeinen jedoch eine viel kürzere Nachricht erwartet, werden die empfangenen Datentoken über den reservierten Speicherbereich hinausgeschrieben und können damit andere Daten im Heap oder sogar den Stack überschreiben, womit der Programmfluß beeinträchtigt werden kann.

- **Nur Zeitfehler**: in dieser Klasse sind alle Fehler aufgelistet, die außer einer zeitlichen Verzögerung bei Daten-, EOP- oder EOM-Token keine weiteren Auswirkungen auf den Bitstrom am Ausgangslink hatten.

Bei den in Abb. 5-3 angegebenen Daten lassen sich die Prozentangaben der Fehlertypen nicht zu *100%* addieren, da es möglich ist, daß im gleichen Fehlerinjektionsexperiment am Ausgabestrom zuerst ein EOP-Token fehlt, dann Datentoken verfälscht werden und nach einigen weiteren Token gar keine Ausgabe mehr erfolgt. Einzig die Klasse Nur Zeitfehler tritt ausschließlich, d.h ohne weitere Fehler im Ausgabestrom auf. Desweiteren beziehen sich die Prozentangaben der Abbildung immer auf die Gesamtmenge der tatsächliche aufgetretenen Fehler.

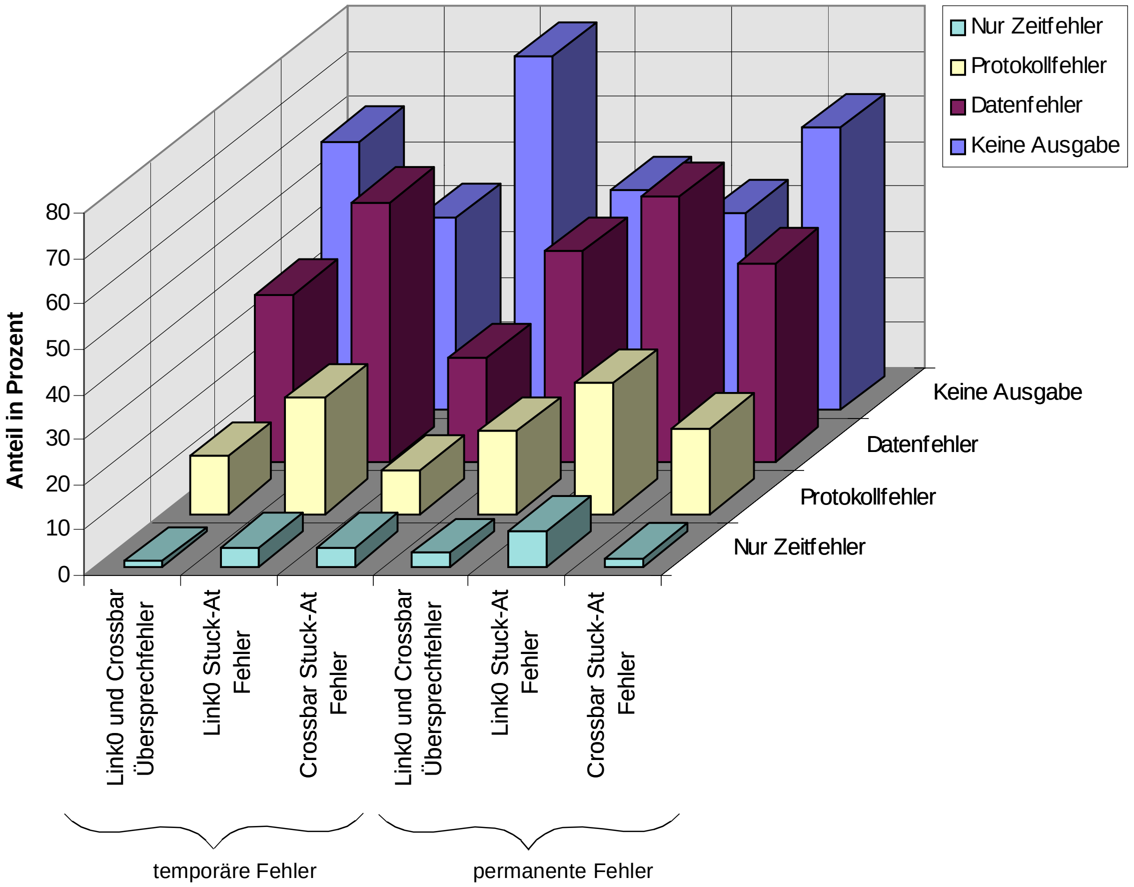


Abb. 5-3: Wahrscheinlichkeit für verschiedene Protokollfehler

In allen sechs Experimentreihen liefern über *40%* der Fehler ab einem gewissen Zeitpunkt der Beobachtung keine Ausgabe mehr. Betrachtet man temporäre Stuck-At Fehler in der

Crossbar-Komponente, so geschieht dies sogar in annähernd *80%* aller Fehlerfälle nach einer Fehlerinjektion. Insgesamt läßt sich für diesen Fehlertyp ablesen, daß er bei Stuck-At Fehlern im *Crossbar* am häufigsten vorkommt, bei Stuck-At-Fehlern in der *Link*-Komponente deutlich seltener und bei den Übersprechfehlern[1] ein Wert zwischen diesen beiden Polen zu erkennen ist. Diese Beobachtung ist unabhängig davon, ob permanente oder temporäre Fehler injiziert wurden. Die erhöhte Wahrscheinlichkeit eines Keine Ausgabe-Fehlers bei Stuck-At-Fehlern in der *Crossbar* Komponente läßt sich darauf zurückführen, daß dort gemäß ihrer Funktionalität die größte Wahrscheinlichkeit einer fehlerhaften Weiterleitung eines Paketes besteht.

Im Gegensatz dazu sind die meisten Daten- und Protokollfehler durch Fehlerinjektionen in der *Link*-Komponente aufgetreten. Dies ist wiederum damit zu erklären, daß sich der Strom der Token (EOP, EOM und Daten) am längsten in der *Link*-Komponente aufhält und damit eine Verfälschung dort wahrscheinlicher ist als im *Crossbar*. Wie bei der Klasse der Keine Ausgabe-Fehler können wir auch hier eine Unabhängigkeit von temporären und permanenten Fehlern erkennen. Insgesamt ist die Wahrscheinlichkeit für Protokoll- und Datenfehler an den Ausgangslinks bei permanenten Fehlern deutlich größer als bei temporären Fehlern. Desweiteren tauchen Datenfehler auch um einen Faktor 2 häufiger auf als Protokollfehler. Die Klasse der Nur Zeitfehler war bei keiner der Experimentreihen nenneswert vertreten. Einzig bei den permanenten Stuck-At Fehlern im *Link* war mit *7,8%* eine erwähnenswerte Häufigkeit zu beobachten.

Zusammenfassend läßt sich sagen, daß diejenigen Auswirkungen interner Fehler, die sich am Ausgangslink beobachten lassen, meist zuerst einige Zeit Daten verfälschen oder gar den Protokollfluß beeinträchtigen (EOP/EOM Token werden beeinflußt) um danach zu einem vollständigen Ausfall des Links zu führen. Durch das gemeinsame Auftreten mehrerer Fehlerklassen muß man dabei auch mit einer erhöhten Wahrscheinlichkeit rechnen, daß mehrere kommunizierende Prozesse auf evtl. mehreren Prozessoren von einem einzigen Fehler innerhalb des Netzwerkes betroffen sind.

Aus den obigen Überlegungen ist ersichtlich, daß besonders die Protokollfehler zu einem schwer diagnostizierbaren Fehlerverhalten führen können, da mehrere Prozesse betroffen sein können, die an der fehlerbehafteten Kommunikation gar nicht teilgenommen haben. Aus diesem Grund soll in der folgenden Abb. 5-4 auf Seite 77 diese Fehlerklasse genauer untersucht werden. Dazu wird die Klasse der Protokollfehler in drei Subklassen aufgeteilt:

- **EOX-over-DATA**: hierbei ist statt eines erwarteten Datentoken ein EOP- oder ein EOM-Token gesendet worden. Das aktuelle Paket ist also unvollständig, was dazu führen kann, daß der Sendeprozeß blockiert bleibt, da er seine folgenden Daten nicht quittiert bekommt. Der Empfangsprozeß kehrt mit einem nicht vollständig erhaltenen Datensatz aus seiner Empfangsroutine zurück. Ein größeres Problem besteht jedoch in dem (regulären) Datentoken, das nach dem fälschlicherweise eingefügten EOP/EOM-Token folgt. Dieses wird von der Hardwareeinheit, die physikalisch mit dem betroffenen Ausgangslink verbunden ist, als Header des darauffolgenden Pakets interpretiert. Durch das Intervalrouting-Schema des STC104 wird der fälschlich interpretierte Header zusammen mit den darauf folgenden Datentoken einem Empfangsprozessor zugestellt, der evtl. gar nichts mit der aktuellen Kommunikation durch den fehlerhaften STC104 zu tun hat. Die damit verbundenen Auswirkungen wurden schon weiter oben in diesem Kapitel beschrieben.

---

[1] Hierbei sei nochmals darauf hingewiesen, daß die Übersprechfehler in beide Komponenten injiziert wurden.

- **DATA-over-EOX**: umgekehrt zum ersten Fall hat man es hier mit einem Datentoken statt eines EOP/EOM-Tokens zu tun. Damit wird das aktuelle Paket verlängert, was zu dem oben ausgeführten Überschreiben des Speichers des Empfangsprozessors führen kann.

- **EOP - EOM**: hierbei wird der Fall betrachtet, daß statt eines EOP-Tokens, welches das Ende eines Pakets kennzeichnet, ein EOM-Token und damit die Endekennung der aktuellen Kommunikation gesendet wird. Desweiteren kann ein EOP-Token statt eines EOM-Tokens gesendet worden sein oder ausschließlich EOP- oder EOM-Tokens (ohne dazwischenliegende Datentoken). Im ersten Fall hat man eine Verkürzung der Nachricht. Das darauf folgende (reguläre) Paket wird erst bei der nächsten Empfangsbereitschaft des Empfangsprozesses quittiert, da die aktuelle Nachricht für den Empfangsprozeß abgeschlossen ist. Im zweiten Fall wird die aktuelle Nachricht verlängert, was wiederum zum Überschreiben des Speichers des Empfangsprozessors führen kann. Sendet der STC104 nur noch EOP/EOM-Token, ist das Verhalten des Empfangsprozessors bzw. des direkt angeschlossenen STC104 nicht spezifiziert und damit unvorhersehbar.

Aus Abb. 5-4 ist ersichtlich, daß bei temporären Stuck-At Fehlern die Klasse EOP-EOM kaum ins Gewicht fällt während sich die anderen beiden Klassen die Waage halten. Bei permanent injizierten Stuck-At Fehlern hat die Klasse EOP-EOM jedoch einen viermal so großen Anteil (*30%* bzw. *40%*) was vor allem zu Lasten von Fehlern der Klasse EOX-over-Daten geht. Im Vergleich zu den temporären Übersprechfehlern ist der Anteil der EOP-EOM Fehler bei permanenten Übersprechfehlern mit *40%* etwa doppelt so groß. Auch hier geht dies vor allem zu Lasten der Klasse EOX-over-Daten.

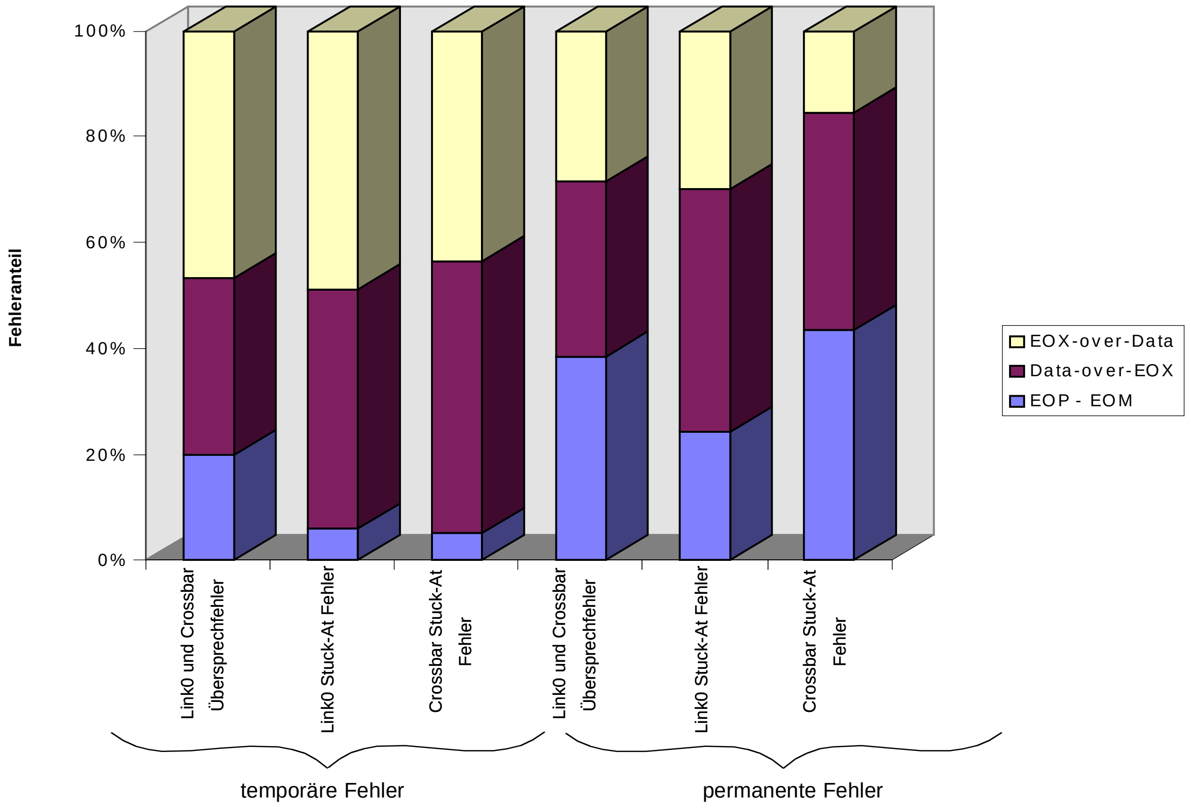


Abb. 5-4: Aufteilung der Protokollfehler beim STC104

Zusammenfassend kann man jedoch bei den Protokollfehlern sagen, daß die gravierenden Fehler, d.h. diejenigen, die Prozesse im Gesamtsystem beeinflussen können, welche mit der fehlerhaften Komponente gar nichts zu tun haben, bei keiner der Untersuchungen vernachlässigt werden können.

5.1.4 Fehlerlatenz beim STC104

Zum Abschluß der Untersuchungen des Protokollfehlerverhaltens am STC104 soll jetzt noch die Latenz der Fehler angegeben werden. Dabei wird unter dem Begriff der Fehlerlatenz in diesem Zusammenhang die Zeitdauer zwischen der Fehlerinjektion und dem Auftreten des ersten Fehlers an einem der Ausgangslinks des STC104 verstanden.

In Abb. 5-5 wurden dazu die Meßwerte der mittleren Latenzzeiten aufgeschlüsselt und nach den einzelnen Fehlermodellen und Fehlerauswirkungen zusammengefaßt. Aus dem Diagramm läßt sich erkennen, daß mit Ausnahme der Daten- und Protokollfehler, die durch permanente Stuck-At Fehler bedingt waren, alle mittleren Fehlerlatenzen zwischen *50 ns* und *250 ns* lagen. Verzögerte Ausgaben, wie sie mit der Klasse Nur Zeitfehler gekennzeichnet sind, liegen dabei deutlich am unteren Rand dieser Bandbreite.

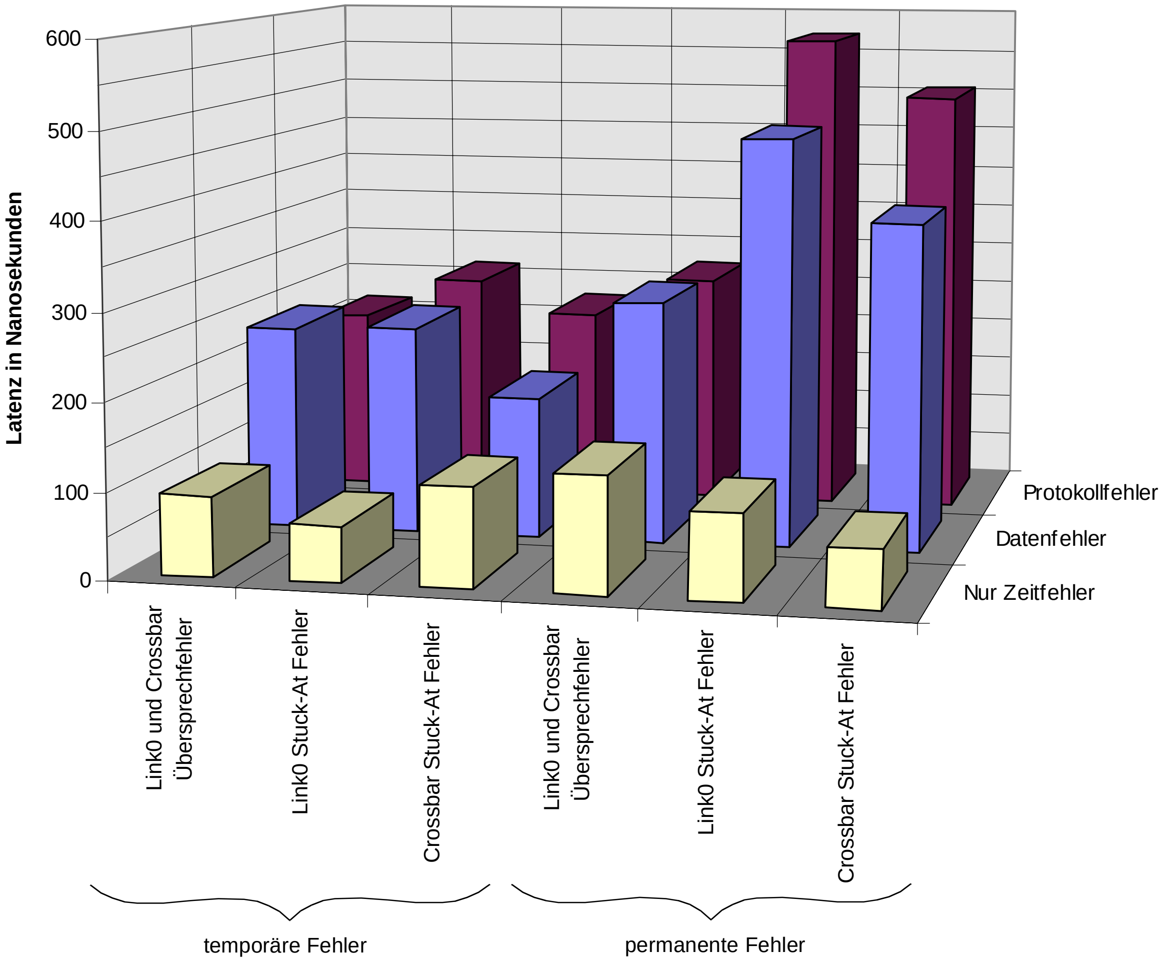


Abb. 5-5: Fehlerlatenz beim STC104 nach Protokollelementen

Einzig die Daten- und Protokollfehler, die von permanenten Stuck-At Fehlern bedingt wurden, weisen eine mittlere Latenzzeit von bis zu *570 ns* auf. Berücksichtigt man dabei jedoch die Designvorgabe des STC104, daß sich ein Daten- bzw. ein Protokolltoken bei freiem Ausgangslink nicht länger als *1 µs* im Switch ist, sind alle mittleren Latenzzeiten in einem erwartungsgemäßen Rahmen. Das heißt insbesondere, daß mit nur einer geringen Wahrscheinlichkeit zu erwarten ist, daß sich ein interner Fehler des STC104 erst nach unverhältnismäßig langer Zeit am Ausgang des Switches bemerkbar macht.

In diesem Zusammenhang soll jedoch darauf hingwiesen werden, daß keine Meßwerte vorliegen, wieviele der injizierten Fehler sich als permanente Fehler im *Interval-Selector* des STC104 manifestiert haben, sich aber bei den geforderten Routingaufgaben des Switches nicht bemerkbar gemacht haben. Die Aktivierung dieser Fehler ist stark abhängig von dem Einsatzgebiet des Switches, d.h. dem Kommunikationsverhalten, der Anzahl und der räumlichen Verteilung der Prozesse, die diesen Switch zu ihrer Aufgabenbewältigung benötigen.

5.2 Protokollverhalten bei Fehlern im Knockout-Switch

Wie auch beim STC104 wurde beim Knockout-Switch jeder der vier modellierten Eingangs- bzw. Ausgangslinks mit einer Instanz des in Kapitel 3.3.2 auf Seite 41 vorgestellten Lastgenerators verbunden. An den Ausgangslinks wird analog zu den Lastgeneratoren des STC104 der eintreffende Datenstrom mitprotokolliert. Bei einem Experimentumfang von jeweils *5000* temporären und permanenten injizierten Fehlern wurde in diesem Fall ein Beobachtungszeitraum von *5000 ns* gewählt. Dabei wurde wiederum der Datenstrom des Golden Run als Referenz herangeogen, um das Verhalten im Fehlerfall zu analysieren.

Im Gegensatz zu den Untersuchungen des Protokollfehlerverhaltens beim STC104 hat man es beim Knockout-Switch mit Fehlern zu tun, die ausschließlich an einem einzelnen Ausgangslink injiziert wurden. Durch die Architektur des Switches ist es ausgeschlossen, daß sich ein Fehler innerhalb eines Ausgangslinks auch an anderen Ausgangslinks bemerkbar machen kann. Als weiterer Unterschied ist hierbei die feste Zellgröße beim ATM-Protokoll zu nennen, die eine Endekennung eines Pakets bzw. einer Nachricht überflüssig macht. Es gibt also kein Äquivalent zu den beim STC104 so wichtigen und fehleranfälligen EOP/EOM-Token. Desweiteren ist es beim ATM-Protokoll erlaubt, daß im Falle einer Überlastsituation ganze Zellen weggeworfen werden dürfen. Die Integrität der übertragenen Daten wird dabei von höheren Schichten des ATM-Protokolls gewährleistet, die zu diesem Zwecke Sequenznummern und eine CRC[1]-Kodierung einsetzen [Pry 93]. Damit erspart man sich beim ATM-Protokoll die Flußkontrolle auf den unteren beiden Schichten des Protokollstacks. Wie bei der Übersicht des ATM-Protokolls in Kapitel 3.2.1.2 auf Seite 33 schon erwähnt wurde, geschieht die Zellsynchronisation ausschließlich über das CRC-Element des Zellheaders.

In der folgenden Abb. 5-6 ist die Wahrscheinlichkeit eines Fehlers am Ausgangslink des Knockout-Switches nach erfolgter Fehlerinjektion getrennt für permanente und temporäre Fehler angegeben. Dabei wurden die Ausgangsfehler gemäß den Hauptprotokollelementen einer Zelle, d.h. Fehler im Header- bzw. im Datenteil, und nach dem Fehlertyp Keine Ausgabe

---

[1] Cyclic Redundancy Checks

aufgeschlüsselt. Dabei zeigt sich wie schon beim STC104, daß selbst bei permanenten Fehlern nur *30%* der injizierten Fehler während des Beobachtungszeitraums zu einem Fehlverhalten am Ausgang des Systems geführt haben. Ebenso analog zum Verhalten beim STC104 weist die Protokollfehlerwahrscheinlichkeit bei temporären Fehlern mit *10%* nur ein drittel des Wertes von permanenten Fehlern auf. Bei einer genaueren Betrachtung der Fehlertypen ist zu erkennen, daß der Fall des Ausfalls des gesamten Ausgangslinks (Keine Ausgabe) mit *1,5%* (permanente Fehler) bzw. mit *0,4%* (temporäre Fehler) sehr gering ist. Dabei ist zu vermerken, daß es in beiden Fällen *4%* der am Ausgangslink beobachteten Fehler sind.

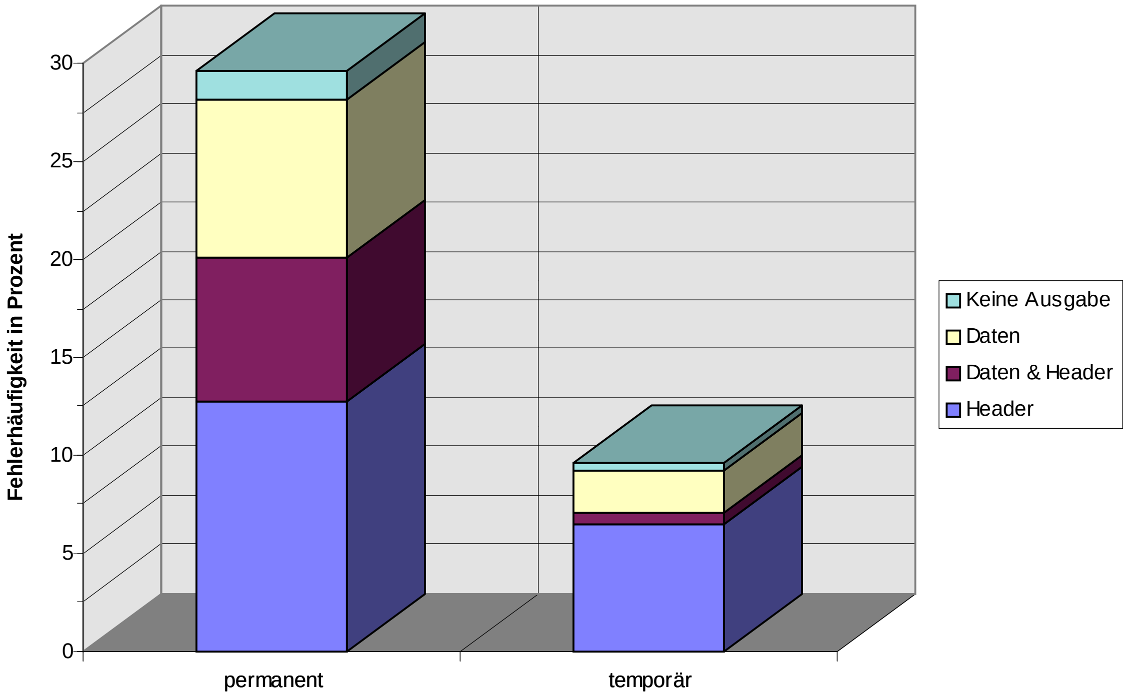


Abb. 5-6: Wahrscheinlichkeit eines Fehlers am ATM-Ausgang nach Fehlerinjektion

Wie zu erwarten ist der Anteil der Fehler, die sich im Daten- und gleichzeitig im Headerteil einer ATM-Zelle manifestieren, bei permanent injizierten Fehlern größer als bei temporär injizierten. Desweiteren ist die Wahrscheinlichkeit eines oder mehrerer Bitfehlers im Header bei temporär injizierten Fehlern etwa dreifach höher als ein Datenfehler. Dies resultiert aus der in Kapitel 3.2.3 auf Seite 37 beschriebenen Entscheidung, nur ein Datenbyte pro ATM-Zelle zu verwenden. Zusammen mit der Designentscheidung, das HEC-Byte erst kurz bevor die ATM-Zelle den Switch verläßt zu berechnen, ergibt sich damit bei temporären Fehlern ungefähr eine Verteilung der Protokollfehler gemäß der Verteilung der Anzahl der Bits in den Protokollelementen.

5.2.1 Fehler in Protokollelementen

Im Gegensatz dazu hat man bei permanenten Fehlern jedoch eine deutlich höhere Wahrscheinlichkeit eines Protokollfehlers der gleichzeitig Daten und Header betrifft, sowie bei reinen Daten und Headerfehlern eine unterschiedliche Aufteilung. Aus diesem Grund soll in diesem

Abschnitt ein detaillierterer Blick auf die Verteilung der Fehler auf die Protokollelemente geworfen werden. Dazu wurde der Header in seine Komponenten VPI, VCI, PTY und CLP zerlegt und der Fehleranteil in diesen Komponenten zusammen mit dem Datenbits jeweils für temporäre und permanent injizierte Fehler ermittelt.

In Abb. 5-7 bezeichnet der mittlere Balken den Anteil des jeweiligen Protokollelements an möglicherweise fehlerbehafteten Bits einer ATM-Zelle[1]. Betrachtet man hier die Verteilung der Protokollelementfehler bei temporär injizierten Fehlern, so ist eine annähernd identische Verteilung wie bei der Bitverteilung zu erkennen. Das läßt darauf schließen, daß sich nur sehr wenige der injizierten Fehler als bleibende Fehler im Kontrollteil des Knockout-Switches manifestieren. Die Kontoll-Logik in Form von FSMs, und damit deren fehlermanifestierenden Registern, spielt also nur eine untergeordnete Rolle bei temporär injizierten Fehlern.

Bei permanent injizierten Fehlern ist die Wahrscheinlichkeit eines Bitfehlers im Header einer ATM-Zelle um so größer desto näher sich das Bit am HEC-Byte des Headers befindet. Dies läßt sich aus der schon weiter oben beschriebenen Tatsache erklären, daß der HEC unbeeinflußt von Fehlern erst direkt am Ausgang des Links berechnet wird und sich permanent injizierte Fehler meist über mehrere Bits erstrecken. Bits im Datenteil der ATM-Zelle haben jedoch eine geringere Wahrscheinlichkeit, von einem permanent injizierten Fehler verfälscht zu werden.

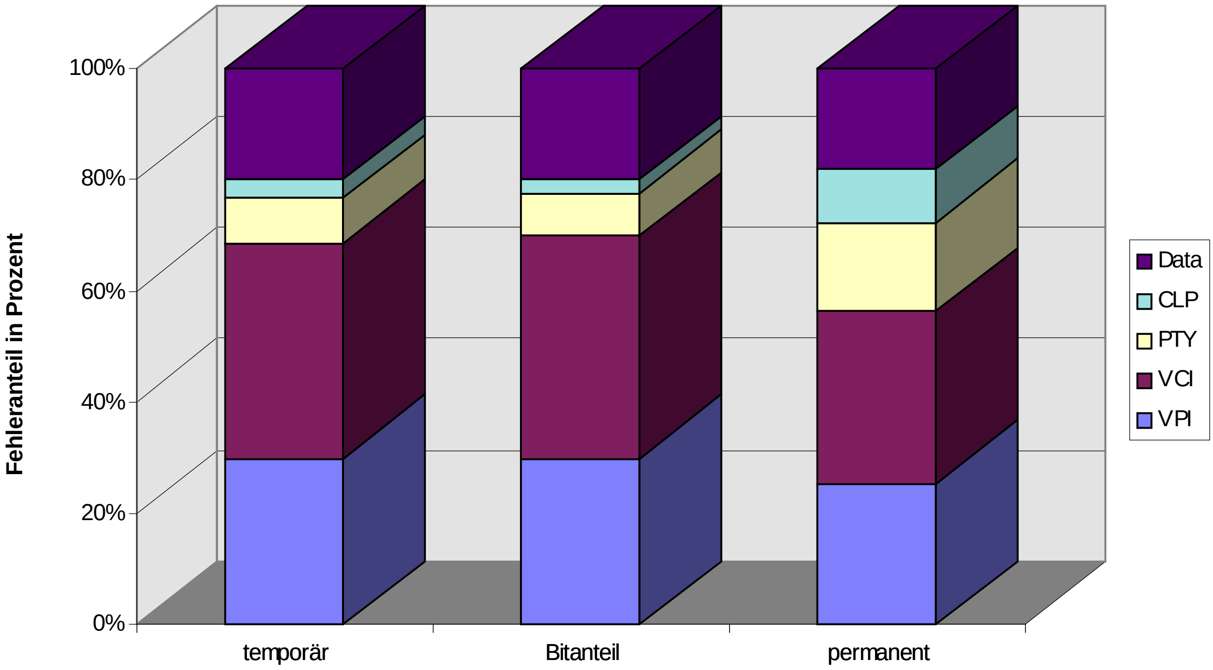


Abb. 5-7: Fehleranteil in den ATM-Protokollelementen

---

[1] Die Daten sind wiederum um die 8 Bit des HEC bereinigt, die keine Fehler enthalten können, da sie erst nach der Fehlerinjektion berechnet werden und somit immer einen korrekten Wert darstellen.

Insgesamt ergibt sich, daß während des Beobachtungszeitraums bei temporär injizierten Fehlern *6%* und bei permanent injizierten Fehlern *20%* zu einem Fehler im Header einer ATM-Zelle führen. Diese Zellen werden im nächsten Switch dann zu einem Prozeß weitergeleitet, der diese Zellen nicht erwartet. Im Gegensatz zum Protokoll des STC104 hat das ATM-Protokoll jedoch wie schon erwähnt eine Flußkontrolle auf den höheren Protokollschichten zwischen den kommunizierenden Prozessen und kann unter anderem an der Sequenznummer auf dieser Ebene die Korrektheit des Zellinhalts erkennen. Zusammen mit dem erlaubten Verlust von gesamten Zellen sind die Fehlerauswirkungen beim STC104 durch dessen Flußkontrolle in Verbindung mit den Endekennungen der Datenpakete variabler Länge gravierender als beim ATM-Protokoll.

5.2.2 Fehlerlatenz beim Knockout-Switch

In der folgenden Abb. 5-8 ist die mittlere Fehlerlatenz aufgetragen, d.h. analog zum STC104 die mittlere Zeit zwischen Fehlerinjektion und der ersten Beobachtung eines Bitfehlers am Ausgangslink. Insgesamt beträgt diese Bandbreite zwischen *1000 ns* und *1550 ns*. Bei einer maximalen Aufenthaltsdauer der Bits im gesamten Ausgangslink von etwa $3\ \mu s$ entsprechen diese Werte den Erwartungen. Insgesamt ist die Latenz bei temporär injizierten Fehlern etwa um *10%* geringer als bei permanent injizierten Fehlern. Eine Erklärung, warum sich diese Fehler früher am Ausgangslink in einem Bitfehler manifestieren ist jedoch nicht bekannt.

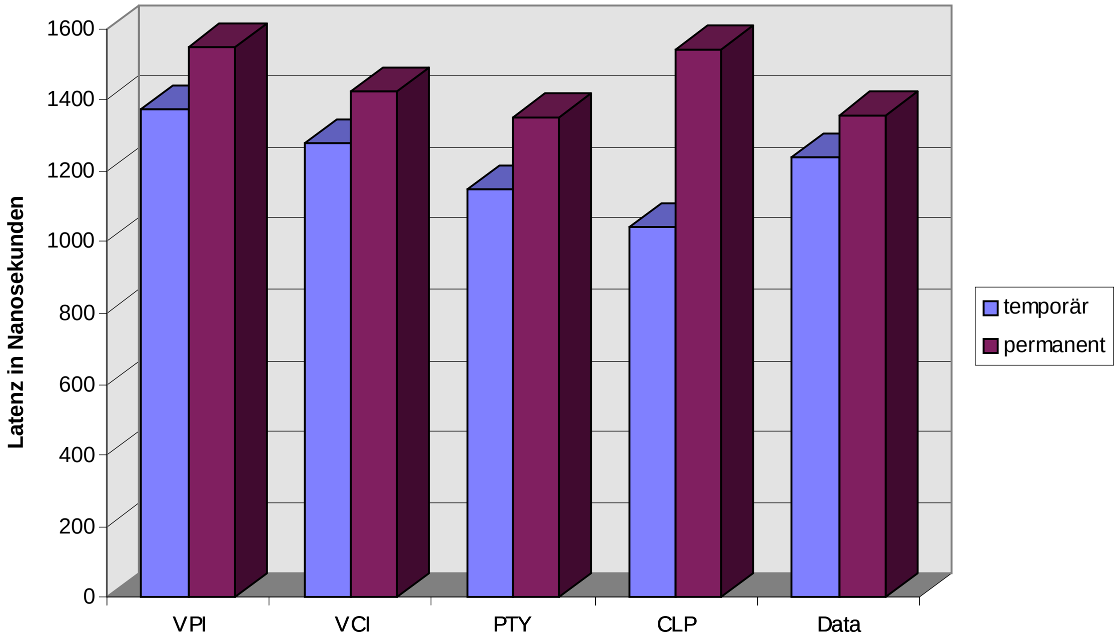


Abb. 5-8: Fehlerlatenz beim ATM-Switch nach Protokollelementen

Bei permanenten wie auch bei temporären Fehlern ist zuerst eine leichte Verringerung der Latenzzeit mit der Nähe zum HEC zu verzeichnen. Bei Fehlern im CLP-Bit ist die Latenz jedoch bei permanent injizierten Fehlern wieder angestiegen. Da es sich bei diesem Protokollelement des Headers um ein einzelnes Bit handelt, liegt die Vermutung nahe, daß die Anzahl aufeinanderfolgender Bitfehler den oben erwähnten Trend überlagert.

5.2.3 Burstiness beim Knockout-Switch

Diese Überlagerung soll anhand der folgenden Abb. 5-9 untersucht werden. Hierbei ist die Burstiness der Bitfehler angegeben, worunter man die Häufigkeit aufeinanderfolgender Bitfehler versteht. Aus der Abbildung läßt sich deutlich erkennen, daß bei temporären Bitfehlern fast immer nur ein einzelnes Bit betroffen ist, während die Bitfehlerhäufigkeiten bei permanent injizierten Fehlern ein weitaus differenzierteres Bild ergeben. Hier steigt die Wahrscheinlichkeit von *N* aufeinanderfolgenden Bitfehlern bis zu *3* Bitfehlern drastisch an und fällt unmittelbar danach auf einen Wert nahe bei *0* ab. Nach einem unregelmäßigen Verlauf ergibt sich erst wieder bei *9*, *12* und *18* aufeinanderfolgenden Bitfehlern eine nennenswerte Häufigkeit ihres Auftretens. Wie bei permanenten Fehlern zu erwarten, ist auch bei Bitfehlern mit mehr als *18* Fehlern mit einer beträchtlichen Häufigkeit zu rechnen.

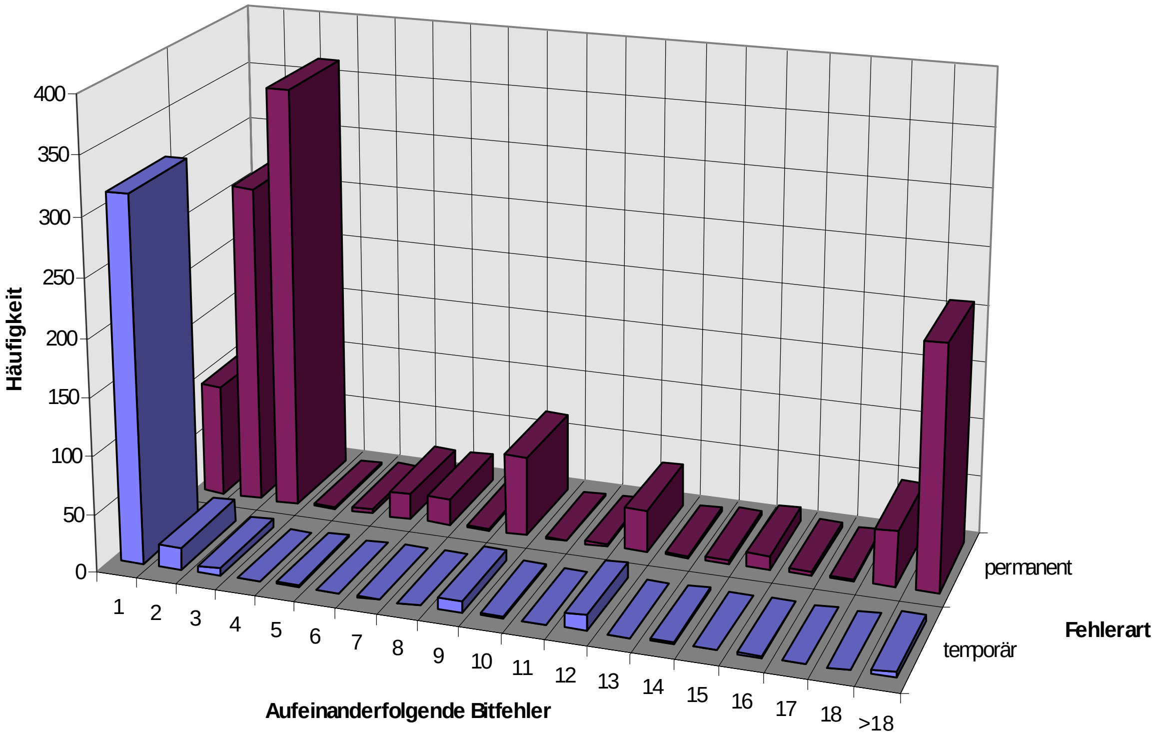


Abb. 5-9: Häufigkeit aufeinanderfolgender Bitfehler

Zusammenfassend läßt sich aus den Meßergebnissen erkennen, daß das ATM-Protokoll bezüglich interner Switch-Fehler weitaus resistenter ist, als das Protokoll des STC104. Dies liegt vor allem in der festen Zellgröße begründet, die eine aufwendige und fehleranfällige Notwendigkeit der Dekodierung und Weiterverarbeitung von Paket- und Nachrichtenendekennungen in den Switches vermeidet.

KAPITEL

6

Bewertung von VERIFY

Dieses Kapitel ist der Bewertung des zur Fehlerinjektion verwendeten Werkzeugs VERIFY gewidmet. Wie bei der Diskussion der Modellanpassungen des STC104 (Kapitel 3.1.3) und des Knockout-Switches (Kapitel 3.2.3) schon ausgeführt wurde, mußte die Größe der beiden Modelle verringert werden, um die Experimente in einer akzeptablen Zeit und mit einem vertretbaren Datenaufkommen durchführen zu können. Dies deutet darauf hin, daß diese beiden Parameter generell bei realen und damit meist sehr großen Systemen zu Problemen führen können. Aus diesem Grund wird in diesem Kapitel zunächst näher auf die Modellgröße und die damit verbundenen Problemen bei der Experimentdurchführung eingegangen.

Die Modellgröße ergab sich im wesentlichen aus dem Anspruch, ein möglichst detailliertes Modell der Hardware zu erstellen, um eine gute Korrelation zum realen Fehlerverhalten der Switches zu erhalten. Zur Fehlermodellierung läßt sich jedoch auch ein Systemmodell auf weit abstrakterer Ebene als der gewählten Gatterebene denken. Aus diesem Grund widmet sich der zweite Teil dieses Kapitels der Untersuchung eines der Switches (Knockout), unter der Einwirkung von Fehlern, die in einem algorithmischen Modell des Switches injiziert wurden.

6.1 Die Modellgröße

Der Wunsch nach einem möglichst genauen System- und Fehlermodell, war eine der Grundlagen dieser Arbeit. Dies wurde erreicht, indem die beiden untersuchten Systeme auf einer sehr hardwarenahen Modellierungsebene, d.h. im vorliegenden Fall auf der Gatterebene modelliert wurden. Der für diesen Detailgrad zu entrichtende Preis war zum einen die enorme Laufzeit der Fehlerinjektionsexperimente, sowie der Bedarf an Plattenplatz, um die Daten abzuspeichern, die über den Beobachtungszeitraum während jeder Fehlerinjektion angefallen sind. Das Ausmaß dieser beiden Parameter soll aus diesem Grund im folgenden verdeutlicht werden.

6.1.1 Einfluß auf die Laufzeit

Alle im folgenden angegebenen Werte basieren auf Messungen, die bei der Experimentdurchführung an einer SUN Sparc Ultra-1, ausgestattet mit 256 MByte RAM ermittelt wurden. Dabei wurde sichergestellt, daß das System keine weiteren benutzerspezifischen Aufgaben außer der VERIFY-Simulation erfüllte. Alle anfallenden Daten wurden auf eine lokale Platte abgelegt, so daß es zu keinen netzwerkbedingten Verzögerungen bei den Fehlerinjektionsexperimenten kommen konnte.

In Tabelle 6-1 sind die Ergebnisse dieser Laufzeitmessungen nach Fehlermodellen zusammengefaßt. Dabei wird neben der benötigten Zeit pro einzelner Fehlerinjektion auch die Dauer eines Gesamtexperiments mit *5000* Fehlerinjektionen inklusive des Golden Run angegeben. Wie bei der Beschreibung von VERIFY in Kapitel 2.2.2 vorgestellt wurde, besteht ein Experiment neben den eigentlichen Simulationen zur Fehlerinjektion auch aus der bei VERIFY integrierten Auswertung der bei jeder Simulation entstehenden Signalspuren. Da sich gezeigt hat, daß diese Auswertung erheblich zur Gesamtzeit eines Fehlerinjektionsexperiments beiträgt, wurde die dafür benötigte Zeit ebenso mit angegeben. Die Experimentparameter entsprechen außer bei permanenten STC104-Fehlern den Parametern, die für die Untersuchungen des Fehlerverhaltens in dieser Arbeit verwendet wurden[1]. Bei permanenten Fehlern im STC104 wurde wie bei temporären Fehlern eine Beobachtungsdauer von *500 ns* gewählt.

		STC104				**Knockout**	
		Stuck-At		**Übersprechfehler**		**Stuck-At**	
		temp	**perm**	**temp**	**perm**	**temp**	**perm**
Simulationsdauer	pro Fehlerinjektion	21 min	20 min	18 min	17 min	10 min	9 min
	5000 Fehlerinjektionen und Golden Run	73,2 Tage	69,7 Tage	62,7 Tage	59,2 Tage	34,8 Tage	31,3 Tage
Dauer Auswertung	pro Fehlerinjektion	107 sec	n.A.	98 sec	n.A.	65 sec	n.A.
	5000 Fehlerinjektionen	148 hrs.	n.A.	136 hrs.	n.A.	90 hrs.	n.A.

Tab. 6-1: Dauer der Experimente

Aus der Tabelle 6-1 ergibt sich für die Experimente beim STC104 eine ungefähr doppelt so hohe Simulationsdauer wie bei den Experimenten am Knockout-Switch. Insgesamt zeigt sich, daß die statistische Notwendigkeit, sehr viele Fehler zu injizieren selbst bei einem sehr leistungsfähigen System zu enormen Simulationszeiten von bis zu *73* Tagen führt. Auch die Auswertezeit von fast *150* Stunden kann dabei nicht vernachlässigt werden. Die etwas geringeren Simulationsdauern bei permanenten Fehlern sind mit der Tatsache zu begründen, daß bei diesen Fehlern keine Signalspuren aufgezeichnet werden mußten. Bei diesen Messungen waren ausschließlich die von dem jeweiligen Switch weitergeleiteten Daten von Interesse, was zu einem geringeren Plattenzugriff geführt hat.

---

[1] Vergl. Kapitel 4.2, Kapitel 4.3 und Kapitel 5.1

Bei der Frage, wie sehr die Wahl des Fehlermodells die Simulationszeit beeinflußt, ist bei den vorliegenden Untersuchungen zu erkennen, daß die Simulation bei Übersprechfehlern um etwa *15%* schneller ist als bei Stuck-At Fehlern. Bei gleichem Detailgrad des Systemmodells ist damit die Simulationsdauer beim komplexeren Fehlermodell geringer. Dies scheint überraschend, da das zu simulierende System mehr Komponenten hat als das Stuck-At Fehlermodell, was durch zusätzlichen Komponenten zur Erzeugung des Übersprechverhaltens bedingt ist. Die kürzere Simulationszeit läßt sich damit erklären, daß im Falle von Übersprechfehlern nicht bei jeder Änderung eines Gatterwertes das mit ihm verbundene Stuck-At Fehlermodell berechnet werden muß.

Hätte man die Designentscheidung, die beiden Systemmodelle um den Faktor *40* zu verkleinern, nicht getroffen, so wäre eine Simulationsdauer von fast *8* Jahren notwendig gewesen[1]. Möchte man VERIFY schon während der Designphase eines digitalen Systems einsetzen, ist eine Dauer von *73* Tagen für ein einzelnes Experiment mit *5000* Fehlerinjektionen bei dem heutigen Druck, möglichst schnell auf dem Markt zu sein, nicht mehr akzeptabel. Eine Lösung dieses Problems ist dabei in der Parallelisierung der Simulationen und der Auswertungen zu finden. Dabei kann ausgenutzt werden, daß die Fehlerinjektionen genauso wie die Auswertungen unabhängig voneinander sind. Ein Experiment mit N Fehlerinjektionen läßt sich also auf k Rechner aufteilen, wobei nur der Golden Run bei allen k Systemen gleich ist. Bezeichne T_s die Simulationszeit für eine Fehlerinjektion, T_a die Auswertungszeit pro Fehlerinjektion und T_g die Dauer des Golden Run. Damit ist leicht ersichtlich, daß man dadurch bei VERIFY den folgenden Speedup S erhält:

$$S = \frac{N \cdot (T_s + T_a) + T_g}{\frac{N}{k} \cdot (T_s + T_a) + T_g}$$

Bei den in dieser Arbeit durchgeführten Experimenten war die Beobachtungszeit nach einer Fehlerinjektion und damit auch T_s und T_a von der Durchlaufzeit der Daten durch die Switches bestimmt. Die Dauer des Golden Run und damit dessen Rechendauer T_g war so gewählt, daß die verschiedensten Protokollelemente von dem jeweiligen Switch behandelt werden mußten. Bedingt durch die Funktion der Switches war die Simulationsdauer des Golden Run um den Faktor *20* größer als die Simulationsdauer einer Fehlerinjektion T_s. Hat man es jedoch mit einem System zu tun, das über einen sehr langen Zeitraum sehr viele unterschiedliche Funktionen zu erfüllen hat, so muß auch die Simulationszeit des Golden Run und damit T_g sehr groß gewählt werden. Besteht bei der Untersuchung dieses Systems jedoch nicht die Notwendigkeit eines sehr langen Beobachtungszeitraums, können die Werte von T_s und T_a um mehrere Größenordungen geringer sein als der Wert von T_g. Hierbei fällt der Speedup bei einer Aufteilung auf k Rechner sehr schlecht aus, da man bei VERIFY auf allen k Rechnern denselben Golden Run berechnet und somit $k - 1$ zwar parallel durchgeführte, jedoch inhaltlich überflüssige Berechnungen des Golden Run erhält. Unter der bestehenden Implementierung von VERIFY lohnt sich ein Verteilen der Experimente auf mehrere Systeme also nur bei einem angemessenen Verhältnis der fehlerfreien Gesamtsimulationszeit zur Beobachtungsdauer. Um

---

[1] Die Simulationsdauer skaliert bei beiden Systemen annähernd linear.

einen Speedup zu erreichen, der unabhängig von diesem Verhältnis ist, müßte VERIFY so verändert werden, daß der Golden Run nur auf einem einzigen System berechnet wird.

In dieser Arbeit wurden je nach Verfügbarkeit bis zu *32* Systeme der Leistungsklasse einer Sparc Ultra-1[1] verwendet sowie bis zu 6 Systeme der Leistungsklasse einer SUN SS20. Der tatsächlich erreichte Speedup bei der Durchführung eines Gesamtexperiments wurde nicht ermittelt, da je nach Verfügbarkeit der Rechner eine unterschiedliche Anzahl an zu simulierenden Fehlerinjektionen pro Rechner verwendet wurde und die verwendeten Rechner goße Laufzeitunterschiede aufwiesen.

6.1.2 Einfluß auf den Speicherplatz

Eine weitere Hürde bei der Durchführung der Experimente stellte der benötigte Speicherplatz dar. Dabei ist dieser Speicherplatz außer bei der statischen Programmgröße unabhängig von den verwendeten Systemen. In der folgenden Tabelle 6-2 werden die Kenngrößen des Speicherplatzbedarfs für die Experimentdurchführung angegeben. Wie bei der Angabe der Experimentdauer werden die Meßwerte auch hier wieder für jedes Fehler- und Systemmodell vorgestellt.

Wie aus der Tabelle 6-2 zu erkennen ist, spielt die statische Programmgröße weder beim Simulationsprogramm (*≤ 1 MByte*), noch beim Auswerteprogramm (ca. *0,2 MByte*) eine große Rolle. Die Komplexität des Modells schlägt sich erst im dynamischen Hauptspeicherplatzbedarf nieder. Hierbei reicht die Spannweite von *9* MByte bei der Simulation des Knockout-Systems bis hin zu *28* MByte bei Stuck-At Fehlern im STC104. Im Gegensatz zur Simulationszeit gibt es beim Hauptspeicherplatzbedarf keinen Unterschied zwischen temporär und permanent injizierten Fehlern. Der größte Bedarf an dynamischem Speicher fällt jedoch nicht bei der Simulation an, sondern bei der Auswertung der Signalspuren. Dieser Wert ist jedoch stark abhängig von der durch die Beobachtungsdauer vorgegebenen Größe der Signalspuren nach einer Fehlerinjektion sowie der beim Golden Run erzeugten Signalspuren.

		STC104				**Knockout**	
		Stuck-At		**Übersprechfehler**		**Stuck-At**	
		temp	**perm**	**temp**	**perm**	**temp**	**perm**
Statische Programmgröße	Simulation	1,0 MByte	1,0 MByte	1,0 MByte	1,0 MByte	0,5 MByte	0,5 MByte
	Auswertung	0,2 MByte	n.A	0,2 MByte	n.A.	0,2 MByte	n.A.
Benötigter Hauptspeicher	Simulation	28 MByte	28 MByte	26 MByte	26 MByte	9 MByte	9 MByte
	Auswertung	68 MByte	n.A.	62 MByte	n.A.	58 MByte	n.A.
Benötigter Plattenplatz	pro Fehlerinjektion	1,0 MByte	n.A.	1,8 MByte	n.A.	2,5 MByte	n.A.
	5000 Fehlerinjektionen	5 GByte	< 1 MByte	9 GByte	< 1 MByte	12,5 GByte	< 1 MByte

Tab. 6-2: Größe der Experimente

---

[1] U.a. DEC-Alpha, HP755, Pentium II mit 300 MHz

Mit einer Spannweite von *58 MByte* bis *68 MByte* ist der benötigte Hauptspeicherplatz jedoch noch im Rahmen von heute üblichen RAM-Größen. Die eigentliche Hürde in diesem Zusammenhang bestand jedoch im benötigten Plattenplatz. Dabei wurde bis zu *12,5 GByte* für ein Experiment von *5000* Fehlerinjektionen benötigt, wobei zu beachten ist, daß im Rahmen dieser Arbeit viele Experimente dieser Größenordnung durchgeführt wurden. Diese Datenmengen konnte nur bewältigt werden, indem die Auswertung schon vor Beendigung aller Fehlerinjektionen eines Experiments gestartet wurde. Dies war möglich, da die Experimente auf mehrere Systeme verteilt wurden (s.o) und somit der zur Auswertung notwendige Golden Run schon vollständig ermittelt war. Der benötigte Plattenspeicherplatz skaliert bei VERFIY linear mit der Beobachtungsdauer und damit der Größe der Signalspurdateien.

Nach der Auswertung der Signalspuren werden diese nicht mehr benötigt und können gelöscht werden. Betrachtet man jedoch den aktuellen Trend hin zu Plattengrößen von über *20 GByte*, so dürfte der Aspekt des benötigten Speicherplatzes bald keine Hürde mehr bei der Erstellung von Fehlerinjektionsexperimenten darstellen.

Die wichtigste Methode, die in VERIFY zur Effizienzsteigerung eingesetzt wurde, ist die in [Sie 98] beschriebene *Multi-Threaded Fault-Injection*. Hierbei wird gewährleistet, daß bei N Fehlerinjektionen nur ein einziges Mal der fehlerfreie Systemzustand während der Gesamtsimulationsdauer berechnet wird. Dies wird dadurch erreicht, daß der fehlerfreie Systemzustand zu einem Fehlerinjektionszeitpunkt t_i eines Fehlers $F_i \quad i \in \{1 \dots N\}$ für alle Fehler F_j mit $t_j \geq t_i$ gleich ist. Der fehlerfreie Ablauf eines Systems muß also nur ein einziges Mal, d.h. bei VERIFY im Golden Run berechnet werden. Ausgehend von diesem fehlerfreien Systemverhalten kopiert VERIFY bei jeder Fehlerinjektion den kompletten Systemzustand des Golden Run und führt die Fehlerinjektion nur auf der Kopie des Golden Run durch. Bezeichne nun T_S die zu simulierende Zeit vom Start der Simulation bis zum eingeschwungenen Zustand, ab dem eine Fehlerinjektion erfolgen soll. Mit T_F werde die mittlere Zeitdauer bezeichnet, die während der Beobachtung einer Fehlerinjektion vergeht. Bei VERIFY besteht T_F aus der mittleren Fehlerinjektionsdauer und dem Beobachtungszeitraum. T_G bezeichne die Gesamtsimulationszeit. Unter den bei VERIFY gewährleisteten Annahmen, daß die Fehlerinjektionszeitpunkte gleichverteilt im Intervall $[T_S, T_G]$ liegen, und die benötigte Zeit für das Kopieren der Systemzustände zu vernachlässigen ist, ergibt sich für die Zeitersparnis der *Multi-Threaded Fault-Injection* ein Faktor von:

$$F \quad = \quad \frac{N \cdot \left(\frac{T_S + T_G}{2} + T_F\right)}{N \cdot T_F + T_G}$$

Im Falle eines 5000 Fehlerinjektionen umfassenden Experiments beim STC104 war $T_F = 1250\ ns$, $T_F = 600\ ns$ und $T_F = 9150\ ns$. Damit ergab sich gemäß obiger Formel eine Beschleunigung um einen Faktor von *2,68* gegenüber der Fehlerinjektion ohne *Multi-Threaded Fault-Injection*. Die Auswirkungen der weiteren Beschleunigungsverfahren in VERIFY konnten nicht quantifiziert werden, da hierfür ein Referenzwerkzeug ohne diese Verfahren notwendig gewesen wäre.

6.2 Abstraktionsebenen des System- und Fehlermodells

Wie sich aus den Betrachtungen der letzten Abschnitte ergeben hat, ist die Experimentdauer selbst bei den schnellsten Prozessoren (bzw. Simulationssystemen) immer noch der am meisten beschränkende Parameter bei der Untersuchung des Fehlerverhaltens realer Systeme. Dies ist vor allem durch den enormen Detailgrad der Modellierung bedingt. Ein naheliegender Ansatz zur Beschleunigung der Experimente bestände also in der Verringerung des Detailgrads. Damit müßten zur Berechnung des Zustands eines zu untersuchenden Systems nicht mehr bis zu mehrere hunderttausend Gatterwerte neu ermittelt werden, sondern nur noch die Werte von abstrakteren Komponenten, deren Verhalten auf algorithmischer Ebene beschrieben ist.

Aus diesem Grund wurde im Rahmen dieser Arbeit auch ein Modell des ATM-Switches erstellt, bei dem die Hauptkomponenten des Switches nur auf algorithmischer Ebene beschrieben wurden. Das dazu gehörige Fehlermodell und die Ergebnisse der damit durchgeführten Fehlerinjektionsexperimente werden im folgenden vorgestellt. Damit soll untersucht werden, ob VERIFY geeignet ist, das Fehlerverhalten bei einer abstrakten Modellierung des Systems zu ermitteln.

6.2.1 System- und Fehlermodell von Komponenten des ATM-Switches

Für die algorithmische Modellierung des Switches wurden neben den für die bisherigen Messungen schon verwendeten Prä- und Postprozessoren auch alle in Kapitel 3.2.2 vorgestellten Komponenten des *Businterfaces* algorithmisch beschrieben [Bog 96]. Bei der Modellierung der Fehler standen dabei mehrere Alternativen zur Auswahl.

Zum einen läßt sich eine Fehlerbeschreibung denken, die sich aus dem Kontrollfluß der Verhaltensbeschreibung der jeweiligen Komponente ergibt. Hierbei gäbe es zum Beispiel für jedes `IF-THEN-ELSE` Konstrukt der Verhaltensbeschreibung die Fehlertypen *Stuck-Then* und *Stuck-Else* [SchA 92]. Im ersten Fall würde die `IF`-Bedingung des Konstrukts während der Fehleraktivierung immer als `TRUE` ausgewertet und somit der `THEN`-Teil des Konstrukts ausgewertet. Analog dazu würde im zweiten Fall immer der `ELSE`-Teil ausgewertet. Die Grundlage dieses Fehlermodells besteht in der Annahme, daß sich ein `IF-THEN-ELSE` Konstrukt auf algorithmischer Ebene in einem Multiplex-Gatter manifestiert, das wiederum ein Stuck-At Verhalten an seinem Ausgang beziehungsweise an seinem Selektoreingang aufweist.

Eine weitere Möglichkeit bestände in einer semantischen Analyse der Verhaltensbeschreibung. Hierbei müßten aus der algorithmischen Implementierung bekannte Hardwarekomponenten abgeleitet werden wie dies auch von Synthesetools wie SYNOPSIS^{TM} bei der automatischen Generierung von Hardwaregattern aus einer algorithmischen Beschreibung geschieht. Für diese Komponenten müßten dann im algorithmischen Modell Fehler gemäß eines dafür schon in einem anderen Umfeld ermittelten Fehlerverhaltens beschrieben und injiziert werden. Beispiele solcher Zuordnungen wären:

- **Zähler**: Zählervariablen sind evtl. auch in der Hardware als Binärzähler implementiert.
- **Speicher**: Felder von Binärwerten stellen Speicherelemente oder Register dar.

- **Zustände**: Zustandsvariablen könnten als Flip-Flops dargestellt werden.
- **Rechenoperationen**: Additionen, etc. sind evtl. durch eine ALU realisiert.

Beide bisher genannten Alternativen basieren jedoch auf der Annahme, daß sich die tatsächliche Hardware-Realisierung eines algorithmischen Beschreibungselements und damit dessen Fehlerverhalten nur aus der abstrakten Beschreibung ermitteln läßt. An dem folgenden Beispiel soll jedoch veranschaulicht werden, daß diese Annahme oft nicht zutrifft. Bei der Realisierung einer Zählervariablen mit Hilfe eines Addierers wäre zum Beispiel eine Rückwärtszählung als Fehlerverhalten nicht ausgeschlossen, während dies bei einem normalen Binärzähler aus Flip-Flops und Gattern nicht vorkommen kann. Es wurde also davon abgesehen, diese sehr hypothetischen Fehlermodelle zu verwenden.

Im Gegensatz dazu kann jedoch davon ausgegangen werden, daß die Ein- und Ausgänge jeder algorithmisch beschriebenen Komponente mit hoher Wahrscheinlichkeit tatsächlichen Signalwerten entsprechen. Hierbei wird für jedes dieser Signale das aus Kapitel 2.3.1 bekannte Stuck-At Fehlermodell angewandt, bei dem der Pegel des Signals für eine gegebene Aktivierungszeit entweder auf den Wert ’1’ (Stuck-At-1) oder auf den Wert ’0’ (Stuck-At-0) gezwungen wird. Durch dieses Fehlermodell wird also eine Pin-Level Fehlerinjektion auf Komponentenebene durchgeführt. Es handelt sich demnach um ein Fehlermodell, das deutlich detaillierter ist, als es bei dem in [ArAA 90] und [MaRS 94] verwendeten Pin-Level Fehlermodell der Fall ist.

Dieses Modell der Pin-Level Fehlerinjektion auf Komponentenebene wurde auch für die folgenden Messungen verwendet, wobei wie bei den bisherigen Untersuchungen auf Gatterebene alle Stuck-At Fehler dieselbe Auftrittshäufigkeit haben. Somit hat jeder Fehler dieselbe Wahrscheinlichkeit, für eine Fehlerinjektion ausgewählt zu werden. Um eine Vergleichbarkeit der Ergebnisse zu gewährleisten, wurde bei der Modellierung algorithmischer Stuck-At Fehler ebenso wie auf der Gatterebene eine Verkleinerung des Nutzdatenanteils einer ATM-Zelle auf ein Byte vorgenommen.

Bei der Modellierung des Systems wurde bei den *Zellfiltern* und jedem der *Konzentratoren*[1] ein Zähler verwendet. Um die eben beschriebene Abhängigkeit von der Implementierung darzustellen, wurden diese beiden Komponenten neben dieser Realisierung auch mit einem externen Aktivitätstrigger implementiert. Dieser Aktivitätstrigger muß dabei nur einmal pro *Businterface* implementiert und das Triggersignal dann mit entsprechenden Verzögerungen an die jeweiligen Komponenten weitergeleitet werden.

6.2.2 Recovery-Verhalten des ATM-Switches auf Algorithmischer Ebene

Für die Darstellung der Ergebnisse aus den Fehlerinjektionsexperimenten wurde dieselbe Diagrammform wie bei der quantitativen Analyse des Ausfallverhaltens verwendet. Eine Erläuterung der Diagrammdarstellung ist in Kapitel 4.1 auf Seite 45 angegeben.

Bei der Auswertung der ersten Messungen mit *7500* injizierten Fehlern zeigte sich eine Problematik, die bei einer Beschreibung auf algorithmischer Ebene typisch ist und in Abb. 6-1

---

[1] Für eine genaue Beschreibung des Aufbaus sei Kapitel 3.2.2 auf verwiesen.

dargestellt wird. Anstatt eines allmählichen Abflachens der Kurve, ist nach *800 ns* ein sprunghafter Anstieg zu verzeichnen, der sich nicht mit dem System- bzw. Fehlermodell erklären läßt. Verdoppelt man den Beobachtungszeitraum von *1000 ns* auf *2000 ns* (hier ohne Abbildung), ist dieser Anstieg nicht mehr im Bereich von *800 ns* zu finden, sondern wandert hin zu *1800 ns* [Bal 96]. Dies läßt darauf schließen, daß das Auswerteprogramm von VERIFY, welches die Signalspuren nach den Fehlerinjektionen mit der Signalspur des Golden Run vergleicht, bei diesem Modell gegen Ende des Beobachtungszeitraums eine Rückkehr zu einem fehlerfreien Zustand ermittelt, obwohl dies nicht dem Systemverhalten entspricht.

Dieses Phänomen läßt sich damit erklären, daß bei einem Verhaltensmodell auf abstrakter Ebene meist Variablen eingesetzt werden. In den Signalspuren, die während des Beobachtungszeitraums nach den Fehlerinjektionen aufgezeichnet werden, sind die Werte dieser Variablen jedoch nicht mit aufgezeichnet. Fehler, die sich in einem falschen Variablenwert manifestieren, müssen jedoch nicht permanent eines der Ausgangssignale der zugehörigen Komponente fälschen. Geschieht diese Auswirkung auf ein oder mehrere Ausgangssignale einer Komponente jedoch nur gelegentlich, so erkennt das Auswerteprogramm speziell am Ende des Beobachtungszeitraums eine Übereinstimmungen mit dem fehlerfreien Golden Run, obwohl sich der Fehler immer noch im System befindet. VERIFY stellt mit der Beschränkung auf die Beobachtung der Signale also keine volle Beobachtbarkeit des Systems zur Verfügung.

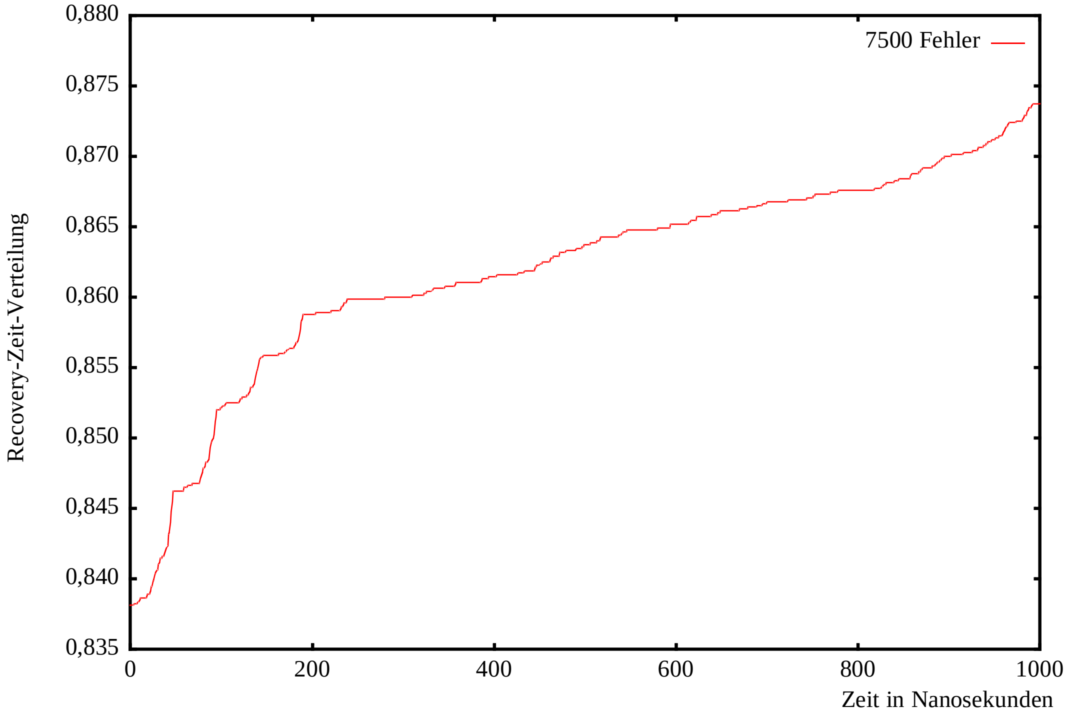


Abb. 6-1: Recovery-Zeiten im algorithmischen ATM-Modell ohne Berücksichtigung interner Zustände

Um dieses Problem zu umgehen, ohne VERIFY zu ändern, wurde die Modellierung jeder Komponente so verändert, daß alle Variablenwerte als Ausgangssignale aus der Komponente herausgeführt sind. Da diese Signale zur eigentlichen Funktionalität des Systems nicht notwendig sind, müssen sie auch nicht mit anderen Komponenten verbunden werden. Sie dienen ausschließlich der Beobachtbarkeit des internen Fehlerverhaltens.

In der folgenden Abb. 6-2 sind die Ergebnisse der Fehlerinjektionsexperimente mit Berücksichtigung der internen Zustände aufgezeichnet. Dabei wurden jeweils *5000* Fehler in das Modell mit replizierten Zählern sowie in das Modell mit Aktivitätstriggern (optimierte Zähler) injiziert. Vergleicht man diese Meßergebnisse mit der Recovery-Zeit-Verteilung, die beim Gattermodell ermittelt wurde (Abb. 4-17 auf Seite 62 und Abb. 4-18 auf Seite 63), so zeigt sich während des gesamten Beobachtungszeitraums ein unterschiedliches Verhalten. Aus diesem Grund wurde in Abb. 6-2 auf eine Untergliederung der Recovery-Zeiten in verschiedene Unterkomponenten verzichtet.

Die Wahrscheinlichkeit, daß ein Fehler nach seiner Aktivierungszeit sofort verschwindet und keinen weiteren Effekt auf das System hat, ist bei Stuck-At Fehlern im algorithmischen Modell um *20%* bzw. *30%* größer als beim Gattermodell. Dies ist dadurch begründet, daß sich die Werte der Ausgangssignale bei algorithmisch beschriebenen Komponenten weit weniger häufig ändern, als dies bei Gattern der Fall ist. Wählt man wie im vorliegenden Fall für beide Modelle den gleichen Wert für die mittlere Fehlerinjektionsdauer, so wird sich während dieser Dauer der Wert eines Gatters mit höherer Wahrscheinlichkeit ändern, als dies bei algorithmisch beschriebenen Komponenten der Fall ist. Man hat es also beim algorithmischen Modell häufiger mit dem Fall zu tun, daß der anliegende Wert durch den Fehler unverändert bleibt.

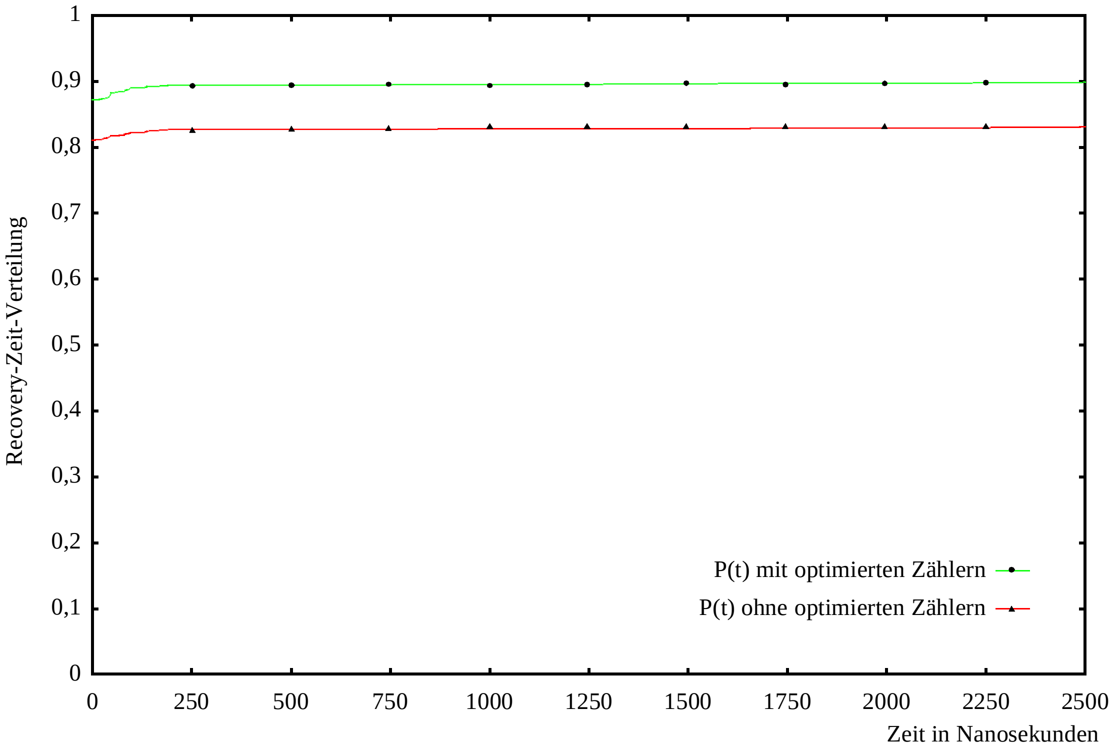


Abb. 6-2: Recovery-Zeiten im algorithmischen ATM-Modell mit Berücksichtigung interner Zustände

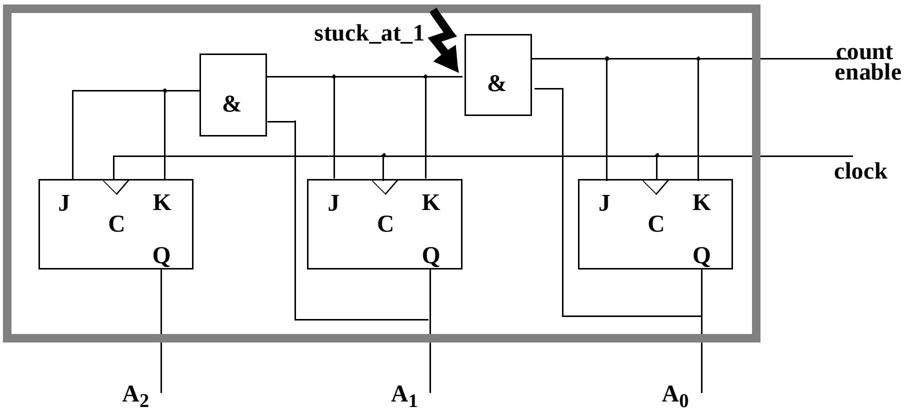
Desweiteren zeigen die Experimente für beide Modellvarianten (replizierter bzw. optimierter Zähler) eine sehr geringe Verbesserung des Anteils behobener Fehler während der ersten *100 ns*. Daraufhin werden fast überhaupt keine weiteren Fehler mehr während der restlichen *2400 ns* des Beobachtungszeitraums behoben. Auch hierbei ist ein gravierender Unterschied zu den Ergebnissen der Experimente am Gattermodell festzustellen, da sich dort während dieses Zeitraums eine Verbesserung um etwa *20%* ergeben hat. Im Gegensatz zum Fehlerverhalten des Gattermodells ist hier also eine nur sehr geringe Dynamik über den Beobachtungszeitraum zu beobachten. Qualitativ verlaufen die beiden Kurven der Modelle mit und ohne optimierten Zähler jedoch gleich. Der eigentliche Unterschied besteht in einem höheren Anteil von Fehlern ohne Auswirkungen auf das System.

Zusammenfassend läßt sich für dieses Fehlermodell sagen, daß es nicht mit dem detaillierteren Modell des auf Gatterebene modellierten Switches korreliert. Obwohl die Simulationsdauer von *73* Tagen auf einen Tag reduziert werden konnte, zahlt sich diese Zeitersparnis wegen der Ungenauigkeit der Ergebnisse nicht aus. Es muß also eine andere Möglichkeit gefunden werden, die Simulation zu beschleunigen.

6.2.3 Qualitative Analyse der analytischen Fehlermodellierung

Im folgenden sollen nun theoretische Überlegungen angestellt werden, die die Modellierung von Fehlern auf algorithmischer Ebene bewerten. Dazu sollen zwei Vorgehensweisen zur Erstellung eines Fehlermodells auf einer abstrakten Ebene untersucht werden. Bei dem heutzutage üblicherweise verwendeten Top-Down Ansatz des Systemdesigns verfeinert man das Systemverhalten während des Designs schrittweise hin zu detaillierteren Komponenten. Will man nun während dieses Prozesses anhand einer Systembeschreibung einer gegebenen Abstaktionsebene dessen Fehlerverhalten ermitteln, so hat man noch kein Wissen über die detaillierte Realisierung der einzelnen Komponenten. Wählt man in dieser Phase ein Fehlermodell für eine gegebene Komponente, so ist dies also noch sehr spekulativ und führt im Allgemeinen bei Fehlersimulationsexperimenten zu einem Systemverhalten, das nicht mit der Realität übereinstimmt. Dies wird in Abb. 6-3 am Beispiel eines 3-Bit Zählers veranschaulicht. Hierbei waren gemäß der Black-Box Betrachtung die Eingangssignale und die Ausgangssignale bekannt. Die interne Realisierung der Komponente wird dabei zunächst nicht betrachtet. Wählt man in diesem Fall ein Pin-Level Fehlermodell, so bekommt man während des Aktivierungszeitraums (grau unterlegter Bereich der Tabelle in Abb. 6-3) die angegebene Folge von Ausgangssignalen und die damit repräsentierten binären Zählerwerte. Betrachtet man jedoch den internen Aufbau der Komponente und wählt auch hier wieder ein Pin-Level Fehlermodell (diesmal an den Ein- und Ausgängen der Gatter) so kann beim angegebenen Stuck-At-1 Fehler eine Zählerfolge an den Ausgangssignalen anliegen, die das Zählerverhalten nachträglich, d.h. auch nach der Aktivierungszeit des Fehlers beeinflußt. Dieser Fehler ist mit dem Pin-Level Fehlermodell der Black-Box prinzipiell nicht modellierbar.

Diese Problematik läßt sich auf alle Fehlermodelle erweitern, die auf einer Black-Box Betrachtungsweise basieren. Nur durch Zufall würde man in diesem Fall ein korrektes, d.h. dem realen Fehlerverhalten entsprechendes Fehlermodell erstellen. Den Ergebnissen von Fehlerinjektionsexperimenten, die auf dieser Vorgehensweise basieren, kann also nicht vertraut werden.



Richtige Zählerfolge	000	001	010	011	100	101	110	111	000	001	010
interner Fehler	000	001	010	101	110	111	000	001	010	011	100
pin-level Fehler	000	001	010	xxx	100	101	110	111	000	001	010

Abb. 6-3: Fehlerauswirkungen bei einem Zähler mit J-K-Flip-Flops

Die einzige Alternative zur korrekten Ermittlung des Fehlerverhaltens eines komplexen Systems auf algorithmischer Ebene besteht also in der umgekehrten Vorgehensweise, d.h. eines Bottom-Up-Designs. Hierbei wird der Designprozeß zuerst so weit vorangetrieben, bis die detaillierteste Beschreibung vorliegt, was im Falle von VHDL dem Gattermodell entspricht. Dort werden nun Fehler gemäß eines gewählten Fehlermodells definiert und damit dann Fehlerinjektionsexperimente im Gattermodell der Komponenten durchgeführt, die in einer höheren Abstraktionsebene der Systembeschreibung definiert sind. Die qualitative und quantitative Analyse des Fehlerverhaltens, die anhand dieser Experimente ermittelt werden kann, fließt dann in die algorithmische Beschreibung dieser Komponenten ein. Somit erhält man schrittweise eine Verhaltensbeschreibung des Systems auf immer abstrakterer Ebene. Das Problem realistischer Eingangssignale für die Komponenten bei der Untersuchung dieser Gattermodelle kann für den Fall einer fehlerfreien Nachbarkomponente die abstrakte Beschreibung des restlichen Systems bilden. Für den Fall, daß eine der Nachbarkomponenten jedoch einen Fehler aufweist, können die Muster der Ausgangssignalwerte jedoch erheblich vom fehlerfreien Fall abweichen. Die vorgeschlagene Vorgehensweise kann das Verhalten einer Komponente bei realistisch fehlerbehafteten Eingangssignalen also nicht berücksichtigen.

Eine Untersuchung der Komponente bezüglich aller möglichen Eingangssignale verbietet sich im Allgemeinen, da bei N Eingangssignalen zu jedem simulierten Zeitpunkt 2^N verschiedene Kombinationen von Eingangssignalwerten untersucht werden müßten. Da reale Komponenten meist mehrere Duzend binäre Eingangssignale besitzen, ist diese vollständige Betrachtung zeitlich nicht sinnvoll. Daneben hätte man Eingangsmuster berücksichtigt, die selbst bei fehlerhaften Nachbarkomponenten nie auftreten könnten.

Für den in dieser Arbeit modellierten STC104 läßt sich anhand der in Kapitel 4.2 und Kapitel 5.1 ermittelten Ergebnisse der Fehlerinjektion ein abstraktes, komplett algorithmisches Modell des STC104 erstellen. Dabei könnten vor allem die Betrachtungen des Kommunikationsprotokolls im Fehlerfall mit in die Verhaltensbeschreibung einfließen. Damit wäre dann der Weg offen, ein komplexes Netzwerk, das aus diesen Komponenten aufgebaut ist, bezüglich eines realistischen Fehlerverhaltens in moderater Zeit zu untersuchen.

KAPITEL 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Mit der vorliegenden Arbeit wurden die Auswirkungen von Hardwarefehlern an zwei Beispielen digitaler Vermittlungseinheiten untersucht. Nach einem Vergleich verschiedener Methoden zur Fehlerinjektion fiel die Wahl auf VERIFY, das anhand einer VHDL-Beschreibung des Systems eine effiziente Fehlerinjektion ermöglicht. Überlegungen zum notwendigen Detailgrad der Modells ergaben dabei, daß es unerläßlich ist, das Fehlerverhalten des Systems auf einer möglichst niedrigen Abstraktionsebene der Systembeschreibung zu untersuchen. Dabei handelt es sich im Falle von VHDL-basiertem Systemdesign um die Beschreibung auf Gatterebene.

Bei VERIFY handelt es sich um einen erst vor kurzem entwickelten Modellierungsansatz, der im Gegensatz zu den meisten anderen Werkzeugen zur Fehlerinjektion die Experimentdurchführung von der Modellerstellung bis hin zur Auswertung unterstützt. Da sich diese Arbeit auf Hardwarefehler digitaler Systeme beschränkt, wurden als zu vergleichende Fehlermodelle temporäre und permanente Stuck-At Fehler sowie ein Übersprechfehlermodell implementiert. Einer der wichtigsten Vorteile von VERIFY besteht in der Konsistenz von Systemmodell und Fehlermodell. Speziell bei der gewählten Systembeschreibung auf Gatterebene bestand das Einbringen des Fehlermodells in einem einfachen Austausch der Gatterbibliothek. Damit konnte die Funktionalität der modellierten Systeme implementiert und getestet werden, ohne auf das Fehlermodell Rücksicht nehmen zu müssen. Einzig beim Modell der Übersprechfehler mußten geringe Ergänzungen des Gattermodells erfolgen, wobei jedoch sichergestellt war, daß auch hier die fehlerfreie Funktionalität und Struktur erhalten blieb.

Bei den beiden modellierten Vermittlungssystemen handelt es sich um den STC104, der ein Kommunikationsprotokoll auf Paketebene implementiert und um einen ATM-Switch, der Datenobjekte einer fest vorgegebene Größe, d.h. Zellen weiterleitet. Während das Kommunikationsprotokoll im ersten Fall eine ausgeklügelte Flußkontrolle auf den unteren Protokollebenen vorschreibt, gibt es bei dem gewählten ATM-Switch keine Flußkontrolle auf den betrachteten

Protokollebenen. Dank VERIFY konnte das Fehlerverhalten innerhalb eines Switches sowie die Auswirkungen auf das jeweilige Kommunikationsprotokoll untersucht werden.

Durch die Wahl der Modellierungsebene war das Gattermodell verschiedener logisch-funktioneller Komponenten wie zum Beispiel Finite-State-Mashines, Vergleichslogiken oder Auswahllogiken vorhanden. Damit war es nun möglich, bei der quantitativen Analyse des Ausfallverhaltens der Systeme nicht nur das Recovery-Verhalten des Gesamtsystems sondern auch das Recovery-Verhalten dieser Komponenten zu untersuchen. Während bei den arithmetisch-logischen Komponenten die selbständige Behebung temporärer Fehler meist sehr kurz nach deren Aktivierungsende geschieht, benötigen Komponenten mit mehreren Registern wesentlich länger für die Systemrecovery. Aus den Messungen hat sich ergeben, daß der Komponentenaufbau einen großen Einfluß sowohl auf die Häufigkeit der Fehler, als auch auf deren Auswirkungen auf das Systemverhalten im Fehlerfall hat. Dabei gibt es jedoch keinen einfachen Indikator, wie zum Beispiel die Anzahl der Register innerhalb einer Komponente, um deren Fehlerverhalten vorherzusagen. Der in der bisherigen Literatur meist verwendete Ansatz, die quantitativen und qualitativen Fehlermerkmale abstrakter Komponenten ohne Wissen über deren tatsächliche Realisierung abzuschätzen, führt demnach zu inkorrekten Ergebnissen.

Beim Vergleich des Stuck-At Fehlermodells mit dem Modell der Übersprechfehler ergab sich bei der Dynamik der Recovery eine bemerkenswerte Ähnlichkeit des Fehlerverhaltens bei den einzelnen Komponenten. Einzig die absoluten Werte der Recovery-Zeit-Verteilung lagen hier bei den Übersprechfehlern höher als bei den Stuck-At Fehlern. Desweiteren haben die dauerhaften Systemfehler bei Übersprechfehlern eine andere Aufteilung auf die Komponenten als beim Stuck-At Fehlermodell. Die Untersuchung des dynamischen Recovery-Verhaltens beim ATM-Switch zeigte, daß vor allem der Zellpuffer am Ausgang eines Links zur Recovery des Gesamtsystems beigetragen hat, während Fehler im restlichen Businterface kaum behoben werden konnten.

Neben den Untersuchungen der Fehlerauswirkung innerhalb der Switches konnten detaillierte Angaben gemacht werden, wie sich die Fehler auf das Verhalten der Kommunikationsprotokolle auswirken. Dieser Teil der Arbeit liefert Hinweise, was bei der Erkennung, Lokalisierung und Behebung von Hardwarefehlern in Netzwerken zu berücksichtigen ist. Hierbei konnte gezeigt werden, daß beim STC104 bis zu *30%* der Systemfehler Auswirkungen auf mehr als einen Ausgangslink hatten. Ein nicht unerheblicher Teil dieser Fehler betraf dabei den protokollspezifischen Teil des Datentransports. Es konnte gezeigt werden, daß diese Protokollfehler in einem Gesamtsystem zu einem nur sehr schwer diagnostizierbaren Fehlerverhalten führen. Bei den Untersuchungen der Fehlerauswirkungen auf das ATM-Protokoll zeigte sich eine Anfälligkeit des Protokollteils des Datenaustausches, was jedoch vor allem durch die Wahl der Implementierungsparameter bedingt war. Insgesamt war dabei zu beobachten, daß das ATM-Protokoll durch den Verzicht auf komplizierte Protokollelemente in den niedrigen Protokollebenen weitaus robuster gegenüber Hardwarefehlern im Switch ist, als dies bei dem flußkontrollbasierten STC104-Protokoll der Fall ist.

7.2 Ausblick

Obwohl das Fehlerinjektionswerkzeug VERIFY eine ganze Reihe an Beschleunigungsmaßnahmen zur Effizienzsteigerung beinhaltet, stellte die statistisch bedingte Notwendigkeit sehr vieler Fehlerinjektionsexperimente eine Hürde bei der Experimentdurchführung dar. Diese bestand in Laufzeiten von bis zu *73* Tagen für ein *5000* Fehlerinjektionen umfassendes Experiment auf einer Sparc Ultra-1. Die reale Dauer eines Fehlerinjektionsexperiments dieser Größe konnte durch Parallelisierung auf etwa zwei Wochen reduziert werden. Da man jedoch bei der Entwicklung von Systemen dieser Größenordnung bedingt durch den Designprozeß mit mehreren Experimentreihen rechnen muß, sind die Kosten derzeit zu hoch für einen praktischen Einsatz.

Ein Ausweg aus dieser Misere liegt in der Ermittlung realistischer Fehlermodelle für abstraktere Beschreibungen der digitalen Systeme. Wie in dieser Arbeit dargestellt wurde, müssen diese Fehlermodelle jedoch anhand von Untersuchungen der detailliertesten Beschreibungsebene ermittelt werden. Jedoch ist es nicht unbedingt notwendig, das Gesamtsystem als Gattermodell zu simulieren. Es wäre denkbar, die Fehlerauswirkungen zuerst auf Komponentenebene zu untersuchen, und in einem zweiten Schritt diese Erkenntnisse in die abstrakte Verhaltensbeschreibung dieser Komponenten einzubringen.

Eine Möglichkeit, die Ergebnisse der Fehlerinjektion auf Gatterebene in eine höhere Beschreibungsebene zu übertragen soll hier speziell für Finite-State-Mashines vorgestellt werden. Hierbei wird meistens eine automatische Übersetzung der algorithmischen Beschreibung in ein Gattermodell mit Hilfe von Synthesetools vorgenommen. Man hat es also mit einer guten Korrelation der beiden Beschreibungsebenen zu tun. Aus den Ergebnissen der Fehlerinjektion am Gattermodell der FSMs läßt sich für jeden Zustand und jeden Eingabewert ein Folgezustand anhand der Belegung der Zustandsregister bestimmen. Für den fehlerfreien Fall erhält man dabei wieder die algorithmische Beschreibung, aus der das Gattermodell synthetisiert wurde. Im Falle injizierter Fehler kämen dabei eventuell noch neue Zustände, falls die ursprüngliche FSM nicht genau eine Zweierpotenz an Zuständen hat, und weitere Übergänge zwischen den Zuständen hinzu. Diese neue FSM repräsentiert dann die ursprüngliche, fehlerfreie Beschreibung und die Beschreibung im Fehlerfall. Die Verhaltensbeschreibung müßte dann um die neuen Zustände und Übergänge erweitert werden. Desweiteren wäre mit diesem Ansatz eine Untersuchung der Komponente mit Hilfe der Methoden des Model-Checking möglich.

Wie schon in der Motivation dieser Arbeit beschrieben wurde, können vor allem die Ergebnisse der Untersuchungen des Protokollverhaltens im Fehlerfall in neu zu entwickelnde Verfahren zur Fehlererkennung, Fehlerlokalisierung und Fehlerbehebung in größeren Netzwerken verwendet werden. Zum Test dieser Verfahren kann ein Netzwerk mehrerer Switches simuliert werden, die jeweils auf einer abstrakten Verhaltensbeschreibung modelliert sind. Diese Verhaltensbeschreibung läßt sich aus den in Kapitel 5 vorgestellten Ergebnissen herleiten, wodurch eine effiziente Untersuchung der Diagnoseverfahren gewährleistet wäre.

ANHANG
A

VHDL-Modell eines Link-Modul Controllers des STC104

```
------------------------------------------------------------------------
-- Name der Komponente: CONTROLLER
--
-- Enthaelt folgende Komponenten:
--      Finite State Machine: Analyse der eintreffenden Datenpakete
--                            Regelung der Flußkontrolle (FCT-Token)
--                            Weiterleitung der Token in die FIFO
--      Token_Counter:        Berechnung des Sendezeitpunkts für das
--                            FCT-Token
--      Flip-Flops und
--      Multiplexer:          Stabilisierung der FSM-Signale zur Daten-
--                            weiterleitung an nachfolgende Komponenten
------------------------------------------------------------------------
library       IEEE;
use           IEEE.std_logic_1164.all;
library       Gates;
use           Gates.gates.all;
use           work.types.all;
---------------------------------------------------

entity controller is
  generic     (tpd: time :=  1 ns);
  port        (
       gnd: in std_logic;

       --Eingaben
       clk_phi1:                     in std_logic;
       clk_phi2:                     in std_logic;

       link_mode_register_in:        in std_logic;
       link_command_register_in:     in data32;
       link_status_register_in:      in data32;

       dva_data_valid:               in std_logic;
       sr_next_token_type:           in std_logic;
       sr_parity_error:              in std_logic;
       sr_data_out:                  in std_logic_vector(0 to 7);
       sr_data_token_flag:           in std_logic;
       sr_control_token_flag:        in std_logic;
       ff_free:                      in std_logic;

       --Ausgaben
       link_status_register_out:     out data32;
       link_status_register_select:  out std_logic;

       sr_parity_range:              out std_logic_vector(0 to 7);
       ff_token_in:                  out std_logic_vector(0 to 7);
       ff_eop_eom_flag:              out std_logic;
       ff_delete_all:                out std_logic;
       ff_next_token:                out std_logic;

       dom_controller_in:            out std_logic;
       dom_reset_in:                 out std_logic;
       dom_fct_received:             out std_logic
  );
end controller;

---------------------------------------------------


architecture structure of controller is

    component controller_fsm
       generic (tpd: time := 1 ns);
       port(
```

```
        clk_phi2:in std_logic;

        --Eingaben
        link_mode_register_in:      in std_logic;
        lc_reset_link:              in std_logic;
        lc_start_link:              in std_logic;
        lc_reset_output:            in std_logic;
        ls_token_received:          in std_logic;

        dva_data_valid:             in std_logic;
        sr_parity_error:            in std_logic;
        sr_data_out_0:              in std_logic;
        sr_data_out_1:              in std_logic;
        sr_control_token_flag:      in std_logic;
        ff_free:                    in std_logic;

        --Ausgaben
        link_status_register_out:          out data32;
        link_status_register_select:       out std_logic;

        sr_parity_range:                   out std_logic_vector(0 to 7);
        parity_error_select:               out std_logic;

        ff_token_in:                       out std_logic_vector(0 to 7);
        ff_delete_all:                     out std_logic;
        ff_next_token:                     out std_logic;

        dom_controller_in:                 out std_logic;
        dom_reset_in:                      out std_logic;
        dom_fct_received:                  out std_logic;

        --Token Counter
        tc_shift_signal:                   out std_logic;
        tc_reset_signal:                   out std_logic;
        tc_full_signal:                    in  std_logic;

        token_latch_signal:                in  std_logic_vector(0 to 7);
        token_latch_select_signal:         out std_logic;

        next_token_type_signal:            in  std_logic;
        next_token_type_select_signal:     out std_logic;

        eop_eom_signal:                    out std_logic;
        eop_eom_select_signal:             out std_logic;

        esc_received_latch_select:         out std_logic;
        esc_received_latch_in:             out std_logic;
        esc_received_latch_out:            in std_logic
    );
end component;

component token_counter
   port (
        gnd:                               in std_logic;
        tc_shift:                          in std_logic;
        tc_reset:                          in std_logic;
        tc_full:                           out std_logic
   );
end component;

-- negiertes clk-Signal (Shift-register benötigt
-- 6 Takte, bis sich sr_parity_error stabilisiert hat)
signal clk_phi2_inv:                       std_logic;
signal clk_phi1_inv:                       std_logic;

-- Signale von und zu token-counter
signal tc_shift_signal, tc_reset_signal:   std_logic;
signal tc_full_signal:                     std_logic;
signal tc_shift, tc_reset :                std_logic;
```

```
  -- Data-Out-Magager Signale
  signal fct_received:                      std_logic;
  signal dom_reset:                         std_logic;
  signal dom_controller :                   std_logic;

  -- Signale fuer Latch, das Token-Typ
  -- zwischenspeichert
  signal next_token_type_signal:            std_logic;
  sginal next_token_type_select_signal:     std_logic;

  -- Signale fuer Latch, das Token des
  -- Shift-Registers zwischenspeichert
  signal token_latch_signal:                std_logic_vector(0 to 7);
  signal token_latch_select_signal:         std_logic;

  -- Signal fuer Stabilisierung des Parity Ranges
  signal parity_error:                      std_logic;
  signal parity_error_select:               std_logic;
  signal control_token_flag:                std_logic;

  --Signale fuer Latch, das EOP/EOM-Flag zwischenspeichert
  signal  eop_eom_select_signal:            std_logic;
  signal  eop_eom_signal:                   std_logic;

  -- Signale fuer esc_received_latch
  signal  esc_received_latch_select:        std_logic;
  signal  esc_received_latch_in:            std_logic;
  signal  esc_received_latch_out:           std_logic;

  signal  sr_par_err_tmp :                  std_logic;
  signal  esc_received_tmp :                std_logic;
  signal  eop_eom_tmp :                     std_logic;
  signal  next_token_type_tmp :             std_logic;
  signal  ff_eop_eom_flag_tmp :             std_logic;

  signal  token_tmp_0, token_tmp_1,
          token_tmp_2, token_tmp_3,
          token_tmp_4, token_tmp_5,
          token_tmp_6, token_tmp_7 :        std_logic;

  signal  lc_reset_link :                   std_logic;
  signal  lc_start_link :                   std_logic;
  signal  lc_reset_output :                 std_logic;
  signal  ls_token_received :               std_logic;
  signal  sr_data_out_0 :                   std_logic;
  signal  sr_data_out_1 :                   std_logic;

  signal  dummy0, dummy1, dummy2,
          dummy3, dummy4, dummy5,
          dummy6, dummy7, dummy8,
          dummy9, dummy10, dummy11,
          dummy12,dummy13, dummy14,
          dummy15, dummy16, dummy17 :       std_logic;


begin
  inv_clk_phi2: inv port map (i => clk_phi2, o => clk_phi2_inv);
  inv_clk_phi1: inv port map (i => clk_phi1, o => clk_phi1_inv);

  lc_reset_link                 <=      link_command_register_in(0);
  lc_start_link                 <=      link_command_register_in(1);
  lc_reset_output               <=      link_command_register_in(2);
  ls_token_received             <=      link_status_register_in(5);
  sr_data_out_0                 <=      sr_data_out(0);
  sr_data_out_1                 <=      sr_data_out(1);
  ff_eop_eom_flag               <=      ff_eop_eom_flag_tmp;

  cfsm: controller_fsm
  port map(
     clk_phi2                   =>      clk_phi2,
     link_mode_register_in      =>      link_mode_register_in,
     lc_reset_link              =>      lc_reset_link,
     lc_start_link              =>      lc_start_link,
     lc_reset_output            =>      lc_reset_output,
     ls_token_received          =>      ls_token_received,
     dva_data_valid             =>      dva_data_valid,
     sr_parity_error            =>      parity_error,
     sr_data_out_0              =>      sr_data_out_0,
     sr_data_out_1              =>      sr_data_out_1,
     sr_control_token_flag      =>      control_token_flag,
     ff_free                    =>      ff_free,
     link_status_register_out   =>      link_status_register_out,
     link_status_register_select=>      link_status_register_select,
     sr_parity_range            =>      sr_parity_range,
     parity_error_select        =>      parity_error_select,
     ff_token_in                =>      ff_token_in,
     ff_delete_all              =>      ff_delete_all,
     ff_next_token              =>      ff_next_token,
     dom_controller_in          =>      dom_controller,
     dom_reset_in               =>      dom_reset,
     dom_fct_received           =>      fct_received,
     tc_shift_signal            =>      tc_shift_signal,
     tc_reset_signal            =>      tc_reset_signal,
     tc_full_signal             =>      tc_full_signal,

     token_latch_signal         =>      token_latch_signal,
     token_latch_select_signal  =>      token_latch_select_signal,
     next_token_type_signal     =>      next_token_type_signal,
     next_token_type_select_signal =>   next_token_type_select_signal,
     eop_eom_signal             =>      eop_eom_signal,
     eop_eom_select_signal      =>      eop_eom_select_signal,
     esc_received_latch_select  =>      esc_received_latch_select,
     esc_received_latch_in      =>      esc_received_latch_in,
     esc_received_latch_out     =>      esc_received_latch_out
  );
```

```
-----------------------------------------------
-- Token Counter
-----------------------------------------------
tc: token_counter
port map (
   gnd            =>    gnd,
   tc_shift       =>    tc_shift,
   tc_reset       =>    tc_reset,
   tc_full        =>    tc_full_signal
);


-----------------------------------------------
-- Latch, um  tc_reset_signal zu stabilisieren
-----------------------------------------------
tc_sh_latch: d_ff
port map(
   clk            =>    clk_phi2_inv,
   d              =>    tc_reset_signal,
   q              =>    tc_reset,
   qn             =>    dummy0
);
-----------------------------------------------
-- Latch, um  tc_shift_signal zu stabilisieren
-----------------------------------------------
tc_rs_latch: d_ff
port map(
   clk            =>    clk_phi2_inv,
   d              =>    tc_shift_signal,
   q              =>    tc_shift,
   qn             =>    dummy1
);

-----------------------------------------------
-- Latch, um  sr_parity_error zu stabilisieren
-----------------------------------------------
sr_control_token_flag_latch: d_ff
port map(
   clk            =>    clk_phi2,
   d              =>    sr_control_token_flag,
   q              =>    control_token_flag,
   qn             =>    dummy2
);

-----------------------------------------------
-- Latch, um  sr_parity_error zu stabilisieren
-----------------------------------------------
sr_parity_error_and: and2
port map(
   i0             =>    sr_parity_error,
   i1             =>    parity_error_select,
   o              =>    sr_par_err_tmp
);
sr_parity_error_latch: d_ff
port map(
   clk            =>    clk_phi1_inv,
   d              =>    sr_par_err_tmp,
   q              =>    parity_error,
   qn             =>    dummy3
);

--------------------------------------------------------
-- Latch, um Token am Shift-Register zwischenzuspeichern
--------------------------------------------------------
token_mux_0: mux2
port map(
   i0             =>    token_latch_signal(0),
   i1             =>    sr_data_out(0),
   sel            =>    token_latch_select_signal,
   o              =>    token_tmp_0
);

token_latch_0: d_ff
port map(
   clk            =>    clk_phi2_inv,
   d              =>    token_tmp_0,
   q              =>    token_latch_signal(0),
   qn             =>    dummy4
);

token_mux_1: mux2
port map(
   i0             =>    token_latch_signal(1),
   i1             =>    sr_data_out(1),
   sel            =>    token_latch_select_signal,
   o              =>    token_tmp_1
);

token_latch_1: d_ff
port map(
   clk            =>    clk_phi2_inv,
   d              =>    token_tmp_1,
   q              =>    token_latch_signal(1),
   qn             =>    dummy5
);

token_mux_2: mux2
port map(
   i0             =>    token_latch_signal(2),
   i1             =>    sr_data_out(2),
   sel            =>    token_latch_select_signal,
   o              =>    token_tmp_2
);
```

```
  token_latch_2: d_ff
  port map(
     clk             =>     clk_phi2_inv,
     d               =>     token_tmp_2,
     q               =>     token_latch_signal(2),
     qn              =>     dummy6
  );

  token_mux_3: mux2
  port map(
     i0              =>     token_latch_signal(3),
     i1              =>     sr_data_out(3),
     sel             =>     token_latch_select_signal,
     o               =>     token_tmp_3
  );

  token_latch_3: d_ff
  port map(
     clk             =>     clk_phi2_inv,
     d               =>     token_tmp_3,
     q               =>     token_latch_signal(3),
     qn              =>     dummy7
  );

  token_mux_4: mux2
  port map(
     i0              =>     token_latch_signal(4),
     i1              =>     sr_data_out(4),
     sel             =>     token_latch_select_signal,
     o               =>     token_tmp_4
  );

  token_latch_4: d_ff
  port map(
     clk             =>     clk_phi2_inv,
     d               =>     token_tmp_4,
     q               =>     token_latch_signal(4),
     qn              =>     dummy8
  );

  token_mux_5: mux2
  port map(
     i0              =>     token_latch_signal(5),
     i1              =>     sr_data_out(5),
     sel             =>     token_latch_select_signal,
     o               =>     token_tmp_5
  );

  token_latch_5: d_ff
  port map(
     clk             =>     clk_phi2_inv,
     d               =>     token_tmp_5,
     q               =>     token_latch_signal(5),
     qn              =>     dummy9
  );

  token_mux_6: mux2
  port map(
     i0              =>     token_latch_signal(6),
     i1              =>     sr_data_out(6),
     sel             =>     token_latch_select_signal,
     o               =>     token_tmp_6
  );

  token_latch_6: d_ff
  port map(
     clk             =>     clk_phi2_inv,
     d               =>     token_tmp_6,
     q               =>     token_latch_signal(6),
     qn              =>     dummy10
  );

  token_mux_7: mux2
  port map(
     i0              =>     token_latch_signal(7),
     i1              =>     sr_data_out(7),
     sel             =>     token_latch_select_signal,
     o               =>     token_tmp_7
  );

  token_latch_7: d_ff
  port map(
     clk             =>     clk_phi2_inv,
     d               =>     token_tmp_7,
     q               =>     token_latch_signal(7),
     qn              =>     dummy11
  );

  ----------------------------------------------
  -- Latch, um dom_ftc_received zu stabilisieren
  ----------------------------------------------
  dom_ftc_recv_latch: d_ff
  port map(
     clk             =>     clk_phi1,
     d               =>     fct_received,
     q               =>     dom_fct_received,
     qn              =>     dummy12
  );

  ----------------------------------------------
  -- Latch, um dom_reset_in zu stabilisieren
  ----------------------------------------------
  dom_reset_latch: d_ff
  port map(
     clk             =>     clk_phi1,
     d               =>     dom_reset,
     q               =>     dom_reset_in,
     qn              =>     dummy13
  );

  ----------------------------------------------
  -- Latch, um dom_controller_in zu stabilisieren
  ----------------------------------------------
  dom_controller_latch: d_ff
  port map(
     clk             =>     clk_phi2_inv,
     d               =>     dom_controller,
     q               =>     dom_controller_in,
     qn              =>     dummy14
  );

  ---------------------------------------------------------
  -- Latch, um Typ des naechsten Tokens zwischenzuspeichern
  ---------------------------------------------------------
  next_token_type_mux: mux2
  port map(
     i0              =>     next_token_type_signal,
     i1              =>     sr_next_token_type,
     sel             =>     next_token_type_select_signal,
     o               =>     next_token_type_tmp
  );

  next_token_type_latch: d_ff
  port map(
     clk             =>     clk_phi2_inv,
     d               =>     next_token_type_tmp,
     q               =>     next_token_type_signal,
     qn              =>     dummy15
  );

  -------------------------------------------------------
  -- Latch, um EOP/EOM-Flag fuer FIFO zwischenzuspeichern
  -------------------------------------------------------
  eop_eom_mux: mux2
  port map(
     i0              =>     ff_eop_eom_flag_tmp,
     i1              =>     eop_eom_signal,
     sel             =>     eop_eom_select_signal,
     o               =>     eop_eom_tmp
  );

  eop_eom_latch: d_ff
  port map(
     clk             =>     clk_phi2_inv,
     d               =>     eop_eom_tmp,
     q               =>     ff_eop_eom_flag_tmp,
     qn              =>     dummy16
  );

  -------------------------------------------------------------
  -- Latch, das anzeigt, dass ein ESC-Token empfangen worden ist
  -------------------------------------------------------------
  esc_received_mux: mux2
  port map(
     i0              =>     esc_received_latch_out,
     i1              =>     esc_received_latch_in,
     sel             =>     esc_received_latch_select,
     o               =>     esc_received_tmp
  );

  esc_received_latch: d_ff
  port map(
     clk             =>     clk_phi2_inv,
     d               =>     esc_received_tmp,
     q               =>     esc_received_latch_out,
     qn              =>     dummy17
  );

end structure;
```

```
--**********************************************************************
--**********************************************************************
--**********************************************************************
--**********************************************************************




----------------------------------------------------------------------
-- controller_fsm
-- Synthetisierbares Verhaltensmodell der Finite State Machine
-- der Controller-Funktionlität
----------------------------------------------------------------------
library     IEEE;
use         IEEE.std_logic_1164.all;
use         work.types.all;
----------------------------------------------------------------------
entity controller_fsm is
  generic (tpd: time := 1 ns);
  port(
     clk_phi2:                     in std_logic;

     --Eingaben
     link_mode_register_in:        in std_logic;
     lc_reset_link:                in std_logic;
     lc_start_link:                in std_logic;
     lc_reset_output:              in std_logic;
     ls_token_received:            in std_logic;

     dva_data_valid:               in std_logic;
     sr_parity_error:              in std_logic;
     sr_data_out_0:                in std_logic;
     sr_data_out_1:                in std_logic;
     sr_control_token_flag:        in std_logic;
     ff_free:                      in std_logic;

     --Ausgaben
     link_status_register_out:     out data32;
     link_status_register_select:  out std_logic;

     sr_parity_range:              out std_logic_vector(0 to 7);
     parity_error_select:          out std_logic;

     ff_token_in:                  out std_logic_vector(0 to 7);
     ff_delete_all:                out std_logic;
     ff_next_token:                out std_logic;

     dom_controller_in:            out std_logic;
     dom_reset_in:                 out std_logic;
     dom_fct_received:             out std_logic;

     --Token Counter
     tc_shift_signal:              out std_logic;
     tc_reset_signal:              out std_logic;
     tc_full_signal:               in  std_logic;

     token_latch_signal:           in  std_logic_vector(0 to 7);
     token_latch_select_signal:    out std_logic;

     next_token_type_signal:       in  std_logic;
     next_token_type_select_signal:out std_logic;

     eop_eom_signal:               out std_logic;
     eop_eom_select_signal:        out std_logic;

     esc_received_latch_select:    out std_logic;
     esc_received_latch_in:        out std_logic;
     esc_received_latch_out:       in std_logic
  );
end controller_fsm;

----------------------------------------------------------------------

architecture behaviour of controller_fsm is

  -- Zustaende fuer den controller_fsm
  type CONTROLLER_FSM_STATE is  (
              DISCONNECTION_ERROR,
              RESET_LINK, RESET_OUTPUT,
              WAIT_FOR_START_LINK,
              SET_LINK_STARTED,
              WAIT_FOR_FCT_0,
              WAIT_FOR_FCT_1,
              CHECK_NEXT_TOKEN_TYPE,
              PARITY_ERROR,
              WAIT_2_0, WAIT_2_1,
              WAIT_1_1, WAIT_1_2,
              WAIT_1_3,
              WAIT_6_0, WAIT_6_1, WAIT_6_2,
              WAIT_6_3, WAIT_6_4, WAIT_6_5,
              GET_TOKENS,
              CHECK_FCT_NUL_OR_EOP_EOM,
              SEND_TYPE_TO_FIFO );


  signal STATE, NEXT_STATE: CONTROLLER_FSM_STATE;
  signal OVER_STATE, OVER_NEXT_STATE: CONTROLLER_FSM_STATE;

begin

  -----------------------
  -- Zustaende umschalten
  -----------------------
```

```
-------------------------------
-- latch fuer naechsten Zustand
-------------------------------
state_latch: process(clk_phi2)
begin
   if clk_phi2'event and clk_phi2 = '1' then
     STATE <= NEXT_STATE after tpd;
   end if;
end process;


-----------------------------------
-- latch fuer uebernaechsten Zustand
-----------------------------------
over_state_latch: process(clk_phi2)
begin
   if clk_phi2'event and clk_phi2 = '1' then
     OVER_STATE <= OVER_NEXT_STATE after tpd;
   end if;
end process;


-----------------------
-- I/O in jedem Zustand
-----------------------

process(  STATE,          lc_reset_link,
          lc_start_link,  lc_reset_output,
          dva_data_valid, ls_token_received)

begin


-----------------------------------------------------------------
-- data-valid, strobe-impuls muss passen!, token_received checken
-----------------------------------------------------------------

 case STATE is

   when RESET_LINK =>
     link_status_register_out <= (others => '0') after tpd;
     link_status_register_select <= '1' after tpd;
     sr_parity_range <= (others => '0') after tpd;
     parity_error_select <= '0' after tpd;
     ff_token_in <= (others => '0') after tpd;
     eop_eom_signal <= '0' after tpd;
     eop_eom_select_signal <= '1' after tpd;
     ff_delete_all <= '1' after tpd;
     ff_next_token <= '0' after tpd;
     dom_controller_in <= '0' after tpd;
     dom_reset_in <= '1' after tpd;
     dom_fct_received <= '0' after tpd;

     esc_received_latch_select <= '1' after tpd;
     esc_received_latch_in <= '0' after tpd;

     -- token-counter ruecksetzen
     tc_shift_signal <= '0' after tpd;
     tc_reset_signal <= '1' after tpd;

     next_token_type_select_signal <= '0' after tpd;
     token_latch_select_signal <= '0' after tpd;

     -------------------------------------------
     -- Restliche Signale, fuer Synthese notwendig
     -------------------------------------------
     --Port-Ausgaenge

     --Signale in der Komponente
     OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value

     -------------------------------------------


     if lc_reset_link = '1' then

        NEXT_STATE <= RESET_LINK after tpd;

     elsif dva_data_valid = '0' and ls_token_received = '1' then

        --localize_error abfragen
        if link_mode_register_in = '1' then
             NEXT_STATE <= RESET_LINK after tpd;
        else
             NEXT_STATE <= DISCONNECTION_ERROR after tpd;
        end if;

     elsif lc_reset_output = '1' then

        NEXT_STATE <= RESET_OUTPUT after tpd;

     else

        NEXT_STATE <= WAIT_FOR_START_LINK after tpd;

     end if;


   when DISCONNECTION_ERROR =>


     link_status_register_select <= '1' after tpd;
     link_status_register_out <= (0 => '1', 4 => '1', others => '0')
                                  after tpd;
```

```
  ---------------------------------------------
  -- Restliche Signale, fuer Synthese notwendig
  ---------------------------------------------
  --Port-Ausgaenge
   sr_parity_range <= (others => ’0’) after tpd;
   parity_error_select <= ’0’ after tpd;
   ff_token_in <= (others => ’0’) after tpd;
   eop_eom_signal <= ’0’ after tpd;
   eop_eom_select_signal <= ’0’ after tpd;
   ff_delete_all <= ’0’ after tpd;
   ff_next_token <= ’0’ after tpd;
   dom_controller_in <= ’0’ after tpd;
   dom_reset_in <= ’0’ after tpd;
   dom_fct_received <= ’0’ after tpd;

  --Signale in der Komponente
   tc_shift_signal  <= ’0’ after tpd;
   tc_reset_signal <= ’0’ after tpd;
   next_token_type_select_signal <= ’0’ after tpd;
   token_latch_select_signal <= ’0’ after tpd;
   esc_received_latch_select <= ’0’ after tpd;
   esc_received_latch_in <= ’0’ after tpd;
   OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
  ---------------------------------------------

  if lc_reset_link = ’1’ then

     NEXT_STATE <= RESET_LINK after tpd;

  elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

     --localize_error abfragen
     if link_mode_register_in = ’1’ then
          NEXT_STATE <= RESET_LINK after tpd;
     else
          NEXT_STATE <= DISCONNECTION_ERROR after tpd;
     end if;

  elsif lc_reset_output = ’1’ then

     NEXT_STATE <= RESET_OUTPUT after tpd;

  else

     NEXT_STATE <= WAIT_FOR_START_LINK after tpd;

  end if;


when RESET_OUTPUT =>
  dom_reset_in <= ’1’ after tpd;
  --reset_output_complete, rueckgesetzt in set_link_started
  link_status_register_select <= ’1’ after tpd;
  link_status_register_out <= (2 => ’1’, others => ’0’) after tpd;

  ---------------------------------------------
  -- Restliche Signale, fuer Synthese notwendig
  ---------------------------------------------
  --Port-Ausgaenge
   sr_parity_range <= (others => ’0’) after tpd;
   parity_error_select <= ’0’ after tpd;
   ff_token_in <= (others => ’0’) after tpd;
   eop_eom_signal <= ’0’ after tpd;
   eop_eom_select_signal <= ’0’ after tpd;
   ff_delete_all <= ’0’ after tpd;
   ff_next_token <= ’0’ after tpd;
   dom_controller_in <= ’0’ after tpd;
   dom_fct_received <= ’0’ after tpd;

  --Signale in der Komponente
   tc_shift_signal  <= ’0’ after tpd;
   tc_reset_signal <= ’0’ after tpd;
   next_token_type_select_signal <= ’0’ after tpd;
   token_latch_select_signal <= ’0’ after tpd;
   esc_received_latch_select <= ’0’ after tpd;
   esc_received_latch_in <= ’0’ after tpd;
   OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
  ---------------------------------------------

  if lc_reset_link = ’1’ then

     NEXT_STATE <= RESET_LINK after tpd;

  elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

     --localize_error abfragen
     if link_mode_register_in = ’1’ then
          NEXT_STATE <= RESET_LINK after tpd;
     else
          NEXT_STATE <= DISCONNECTION_ERROR after tpd;
     end if;

  elsif lc_reset_output = ’1’ then

     NEXT_STATE <= RESET_OUTPUT after tpd;

  else

     NEXT_STATE <= WAIT_FOR_START_LINK after tpd;

  end if;
```

```
when WAIT_FOR_START_LINK =>

   --reset_link
   ff_delete_all <= ’0’ after tpd;
   dom_reset_in <= ’0’ after tpd;

   link_status_register_select <= ’0’ after tpd;
   tc_reset_signal <= ’0’ after tpd;

   --reset_output
   dom_reset_in <= ’0’ after tpd;
   dom_fct_received <= ’0’ after tpd;
   dom_controller_in <= ’0’ after tpd;

   esc_received_latch_select <= ’0’ after tpd;

  ---------------------------------------------
  -- Restliche Signale, fuer Synthese notwendig
  ---------------------------------------------
  --Port-Ausgaenge
   link_status_register_out <= (others => ’0’) after tpd;
   sr_parity_range <= (others => ’0’) after tpd;
   parity_error_select <= ’0’ after tpd;
   ff_token_in <= (others => ’0’) after tpd;
   eop_eom_signal <= ’0’ after tpd;
   eop_eom_select_signal <= ’0’ after tpd;
   ff_next_token <= ’0’ after tpd;

  --Signale in der Komponente
   tc_shift_signal  <= ’0’ after tpd;
   next_token_type_select_signal <= ’0’ after tpd;
   token_latch_select_signal <= ’0’ after tpd;
   esc_received_latch_in <= ’0’ after tpd;
   OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
  ---------------------------------------------


  if lc_reset_link = ’1’ then

     NEXT_STATE <= RESET_LINK after tpd;

  elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

     --localize_error abfragen
     if link_mode_register_in = ’1’ then
          NEXT_STATE <= RESET_LINK after tpd;
     else
          NEXT_STATE <= DISCONNECTION_ERROR after tpd;
     end if;

  elsif lc_reset_output = ’1’ then

     NEXT_STATE <= RESET_OUTPUT after tpd;

  elsif lc_start_link = ’1’ then

     NEXT_STATE <= SET_LINK_STARTED after tpd;

  else

     NEXT_STATE <= WAIT_FOR_START_LINK after tpd;

  end if;


when SET_LINK_STARTED =>
  -- link_started = 1 und reset_output_complete = 0
  link_status_register_select <= ’1’ after tpd;
  link_status_register_out <= (1 => ’1’, others => ’0’) after tpd;

  --schicke FCTs nach aussen
  dom_controller_in <= ’1’ after tpd;

  ---------------------------------------------
  -- Restliche Signale, fuer Synthese notwendig
  ---------------------------------------------
  --Port-Ausgaenge
   sr_parity_range <= (others => ’0’) after tpd;
   parity_error_select <= ’0’ after tpd;
   ff_token_in <= (others => ’0’) after tpd;
   eop_eom_signal <= ’0’ after tpd;
   eop_eom_select_signal <= ’0’ after tpd;
   ff_delete_all <= ’0’ after tpd;
   ff_next_token <= ’0’ after tpd;
   dom_reset_in <= ’0’ after tpd;
   dom_fct_received <= ’0’ after tpd;

  --Signale in der Komponente
   tc_shift_signal  <= ’0’ after tpd;
   tc_reset_signal <= ’0’ after tpd;
   next_token_type_select_signal <= ’0’ after tpd;
   token_latch_select_signal <= ’0’ after tpd;
   esc_received_latch_select <= ’0’ after tpd;
   esc_received_latch_in <= ’0’ after tpd;
   OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
  ---------------------------------------------


  if lc_reset_link = ’1’ then

     NEXT_STATE <= RESET_LINK after tpd;

  elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

     --localize_error abfragen
```

```
      if link_mode_register_in = ’1’ then
          NEXT_STATE <= RESET_LINK after tpd;
      else
          NEXT_STATE <= DISCONNECTION_ERROR after tpd;
      end if;

   elsif lc_reset_output = ’1’ then

      NEXT_STATE <= RESET_OUTPUT after tpd;

   else

      NEXT_STATE <= WAIT_FOR_FCT_0 after tpd;

   end if;


 when WAIT_FOR_FCT_0 =>
   --ruecksetzen
   link_status_register_select <= ’0’ after tpd;

   ---------------------------------------------
   -- Restliche Signale, fuer Synthese notwendig
   ---------------------------------------------
   --Port-Ausgaenge
    link_status_register_out <= (others => ’0’) after tpd;
    sr_parity_range <= (others => ’0’) after tpd;
    parity_error_select <= ’0’ after tpd;
    ff_token_in <= (others => ’0’) after tpd;
    eop_eom_signal <= ’0’ after tpd;
    eop_eom_select_signal <= ’0’ after tpd;
    ff_delete_all <= ’0’ after tpd;
    ff_next_token <= ’0’ after tpd;
    dom_reset_in <= ’0’ after tpd;

   --Signale in der Komponente
    tc_shift_signal  <= ’0’ after tpd;
    tc_reset_signal <= ’0’ after tpd;
    token_latch_select_signal <= ’0’ after tpd;
    esc_received_latch_select <= ’0’ after tpd;
    esc_received_latch_in <= ’0’ after tpd;
    OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
   ---------------------------------------------

   if lc_reset_link = ’1’ then

      next_token_type_select_signal <= ’0’ after tpd;
      dom_controller_in <= ’0’ after tpd;
      dom_fct_received <= ’0’ after tpd;
      NEXT_STATE <= RESET_LINK after tpd;

   elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

      next_token_type_select_signal <= ’0’ after tpd;
      dom_controller_in <= ’0’ after tpd;
      dom_fct_received <= ’0’ after tpd;

      --localize_error abfragen
      if link_mode_register_in = ’1’ then
          NEXT_STATE <= RESET_LINK after tpd;
      else
          NEXT_STATE <= DISCONNECTION_ERROR after tpd;
      end if;

    elsif lc_reset_output = ’1’ then

      next_token_type_select_signal <= ’0’ after tpd;
      dom_controller_in <= ’0’ after tpd;
      dom_fct_received <= ’0’ after tpd;
      NEXT_STATE <= RESET_OUTPUT after tpd;

   elsif sr_data_out_0 = ’0’ and

      sr_data_out_1 = ’0’ and
      sr_control_token_flag = ’1’ then
      -- typ des naechsten token speichern
      next_token_type_select_signal <= ’1’ after tpd;

      -- keine FCTs nach aussen mehr schicken
      dom_controller_in <= ’0’ after tpd;

      -- zeige an, dass FCT empfangen
      dom_fct_received <= ’1’ after tpd;

      NEXT_STATE <= WAIT_2_0;

   else

      --Gegenteil von if
      next_token_type_select_signal <= ’0’ after tpd;
      dom_controller_in <= ’1’ after tpd;
      dom_fct_received <= ’0’ after tpd;

      NEXT_STATE <= WAIT_FOR_FCT_1 after tpd;

   end if;


 when WAIT_FOR_FCT_1 =>
   --ruecksetzen
   link_status_register_select <= ’0’ after tpd;

   ---------------------------------------------
   -- Restliche Signale, fuer Synthese notwendig
   ---------------------------------------------
```

```
   --Port-Ausgaenge
    link_status_register_out <= (others => ’0’) after tpd;
    sr_parity_range <= (others => ’0’) after tpd;
    parity_error_select <= ’0’ after tpd;
    ff_token_in <= (others => ’0’) after tpd;
    eop_eom_signal <= ’0’ after tpd;
    eop_eom_select_signal <= ’0’ after tpd;
    ff_delete_all <= ’0’ after tpd;
    ff_next_token <= ’0’ after tpd;
    dom_reset_in <= ’0’ after tpd;

   --Signale in der Komponente
    tc_shift_signal  <= ’0’ after tpd;
    tc_reset_signal <= ’0’ after tpd;
    token_latch_select_signal <= ’0’ after tpd;
    esc_received_latch_select <= ’0’ after tpd;
    esc_received_latch_in <= ’0’ after tpd;
    OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
   ---------------------------------------------

   if lc_reset_link = ’1’ then

      next_token_type_select_signal <= ’0’ after tpd;
      dom_controller_in <= ’0’ after tpd;
      dom_fct_received <= ’0’ after tpd;
      NEXT_STATE <= RESET_LINK after tpd;

   elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

      next_token_type_select_signal <= ’0’ after tpd;
      dom_controller_in <= ’0’ after tpd;
      dom_fct_received <= ’0’ after tpd;

      --localize_error abfragen
      if link_mode_register_in = ’1’ then
          NEXT_STATE <= RESET_LINK after tpd;
      else
          NEXT_STATE <= DISCONNECTION_ERROR after tpd;
      end if;

   elsif lc_reset_output = ’1’ then

      next_token_type_select_signal <= ’0’ after tpd;
      dom_controller_in <= ’0’ after tpd;
      dom_fct_received <= ’0’ after tpd;
      NEXT_STATE <= RESET_OUTPUT after tpd;

   elsif   sr_data_out_0 = ’0’ and
           sr_data_out_1 = ’0’ and
           sr_control_token_flag = ’1’ then

      -- typ des naechsten token speichern
      next_token_type_select_signal <= ’1’ after tpd;

      -- keine FCTs nach aussen mehr schicken
      dom_controller_in <= ’0’ after tpd;

      -- zeige an, dass FCT empfangen
      dom_fct_received <= ’1’ after tpd;


      NEXT_STATE <= WAIT_2_0;

   else

      --Gegenteil von if
      next_token_type_select_signal <= ’0’ after tpd;
      dom_controller_in <= ’1’ after tpd;
      dom_fct_received <= ’0’ after tpd;

      NEXT_STATE <= WAIT_FOR_FCT_0 after tpd;

   end if;



 when WAIT_2_0 =>
   --token_received setzen
   link_status_register_select <= ’1’ after tpd;
   link_status_register_out <= (1 => ’1’, 5 => ’1’, others => ’0’)
                               after tpd;

   ---------------------------------------------
   -- Restliche Signale, fuer Synthese notwendig
   ---------------------------------------------
   --Port-Ausgaenge
    sr_parity_range <= (others => ’0’) after tpd;
    parity_error_select <= ’0’ after tpd;
    ff_token_in <= (others => ’0’) after tpd;
    eop_eom_signal <= ’0’ after tpd;
    eop_eom_select_signal <= ’0’ after tpd;
    ff_delete_all <= ’0’ after tpd;
    ff_next_token <= ’0’ after tpd;
    dom_controller_in <= ’0’ after tpd;
    dom_reset_in <= ’0’ after tpd;
    dom_fct_received <= ’0’ after tpd;

   --Signale in der Komponente
    tc_shift_signal  <= ’0’ after tpd;
    tc_reset_signal <= ’0’ after tpd;
    next_token_type_select_signal <= ’0’ after tpd;
    token_latch_select_signal <= ’0’ after tpd;
    esc_received_latch_select <= ’0’ after tpd;
    esc_received_latch_in <= ’0’ after tpd;
    OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
```

```
  ---------------------------------------------

  if lc_reset_link = ’1’ then

       NEXT_STATE <= RESET_LINK after tpd;

  elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

       --localize_error abfragen
       if link_mode_register_in = ’1’ then
         NEXT_STATE <= RESET_LINK after tpd;
       else
         NEXT_STATE <= DISCONNECTION_ERROR after tpd;
       end if;

  elsif lc_reset_output = ’1’ then

       NEXT_STATE <= RESET_OUTPUT after tpd;

  else

       NEXT_STATE <= WAIT_2_1 after tpd;

  end if;


when WAIT_2_1 =>

 --------------------------------------------
 -- Restliche Signale, fuer Synthese notwendig
 --------------------------------------------
 --Port-Ausgaenge
   link_status_register_out <= (others => ’0’) after tpd;
   link_status_register_select <= ’0’ after tpd;
   sr_parity_range <= (others => ’0’) after tpd;
   parity_error_select <= ’0’ after tpd;
   ff_token_in <= (others => ’0’) after tpd;
   eop_eom_signal <= ’0’ after tpd;
   eop_eom_select_signal <= ’0’ after tpd;
   ff_delete_all <= ’0’ after tpd;
   ff_next_token <= ’0’ after tpd;
   dom_controller_in <= ’0’ after tpd;
   dom_reset_in <= ’0’ after tpd;
   dom_fct_received <= ’0’ after tpd;

  --Signale in der Komponente
   tc_shift_signal  <= ’0’ after tpd;
   tc_reset_signal <= ’0’ after tpd;
   next_token_type_select_signal <= ’0’ after tpd;
   token_latch_select_signal <= ’0’ after tpd;
   esc_received_latch_select <= ’0’ after tpd;
   esc_received_latch_in <= ’0’ after tpd;
   OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
  --------------------------------------------

  if lc_reset_link = ’1’ then

     NEXT_STATE <= RESET_LINK after tpd;

  elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

     --localize_error abfragen
     if link_mode_register_in = ’1’ then
         NEXT_STATE <= RESET_LINK after tpd;
     else
         NEXT_STATE <= DISCONNECTION_ERROR after tpd;
     end if;

  elsif lc_reset_output = ’1’ then

     NEXT_STATE <= RESET_OUTPUT after tpd;

  else

     NEXT_STATE <= CHECK_NEXT_TOKEN_TYPE after tpd;

  end if;


when CHECK_NEXT_TOKEN_TYPE =>
  --------------------------------------------
  -- Restliche Signale, fuer Synthese notwendig
  --------------------------------------------
  --Port-Ausgaenge
   link_status_register_out <= (others => ’0’) after tpd;
   link_status_register_select <= ’0’ after tpd;
   ff_token_in <= (others => ’0’) after tpd;
   ff_delete_all <= ’0’ after tpd;
   ff_next_token <= ’0’ after tpd;
   dom_controller_in <= ’0’ after tpd;
   dom_reset_in <= ’0’ after tpd;
   dom_fct_received <= ’0’ after tpd;

  --Signale in der Komponente
   tc_shift_signal  <= ’0’ after tpd;
   tc_reset_signal <= ’0’ after tpd;
   next_token_type_select_signal <= ’0’ after tpd;
   token_latch_select_signal <= ’0’ after tpd;
   esc_received_latch_select <= ’0’ after tpd;
   esc_received_latch_in <= ’0’ after tpd;
  --------------------------------------------

  if lc_reset_link = ’1’ then

     eop_eom_signal <= ’0’ after tpd;
```

```
     eop_eom_select_signal <= ’0’ after tpd;
     sr_parity_range <= (others => ’0’) after tpd;
     parity_error_select <= ’0’ after tpd;
     NEXT_STATE <= RESET_LINK after tpd;
     OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value

  elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

     eop_eom_signal <= ’0’ after tpd;
     eop_eom_select_signal <= ’0’ after tpd;
     sr_parity_range <= (others => ’0’) after tpd;
     parity_error_select <= ’0’ after tpd;
     OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value

     --localize_error abfragen
     if link_mode_register_in = ’1’ then
         NEXT_STATE <= RESET_LINK after tpd;
     else
         NEXT_STATE <= DISCONNECTION_ERROR after tpd;
     end if;

  elsif lc_reset_output = ’1’ then

     eop_eom_signal <= ’0’ after tpd;
     eop_eom_select_signal <= ’0’ after tpd;
     sr_parity_range <= (others => ’0’) after tpd;
     parity_error_select <= ’0’ after tpd;
     OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
     NEXT_STATE <= RESET_OUTPUT after tpd;

  elsif next_token_type_signal = ’1’ then

     eop_eom_signal <= ’1’ after tpd;
     eop_eom_select_signal <= ’1’ after tpd;
     sr_parity_range <= b“1100_0000“ after tpd;
     parity_error_select <= ’1’ after tpd;
     OVER_NEXT_STATE <= CHECK_FCT_NUL_OR_EOP_EOM;
     NEXT_STATE <= GET_TOKENS after tpd;

  else

     --Gegenteil von if
     eop_eom_signal <= ’0’ after tpd;
     eop_eom_select_signal <= ’1’ after tpd;
     sr_parity_range <= (others => ’0’) after tpd;
     parity_error_select <= ’0’ after tpd;
     OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
     NEXT_STATE <= WAIT_6_0 after tpd;

  end if;


when WAIT_1_1 =>
  --------------------------------------------
  -- Restliche Signale, fuer Synthese notwendig
  --------------------------------------------
  --Port-Ausgaenge
   link_status_register_out <= (others => ’0’) after tpd;
   link_status_register_select <= ’0’ after tpd;
   sr_parity_range <= (others => ’0’) after tpd;
   parity_error_select <= ’0’ after tpd;
   ff_token_in <= (others => ’0’) after tpd;
   eop_eom_signal <= ’0’ after tpd;
   eop_eom_select_signal <= ’0’ after tpd;
   ff_delete_all <= ’0’ after tpd;
   ff_next_token <= ’0’ after tpd;
   dom_controller_in <= ’0’ after tpd;
   dom_reset_in <= ’0’ after tpd;
   dom_fct_received <= ’0’ after tpd;

  --Signale in der Komponente
   tc_shift_signal  <= ’0’ after tpd;
   tc_reset_signal <= ’0’ after tpd;
   next_token_type_select_signal <= ’0’ after tpd;
   token_latch_select_signal <= ’0’ after tpd;
   esc_received_latch_select <= ’0’ after tpd;
   esc_received_latch_in <= ’0’ after tpd;
  OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
  --------------------------------------------

  if lc_reset_link = ’1’ then

     NEXT_STATE <= RESET_LINK after tpd;

  elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

     --localize_error abfragen
     if link_mode_register_in = ’1’ then
         NEXT_STATE <= RESET_LINK after tpd;
     else
         NEXT_STATE <= DISCONNECTION_ERROR after tpd;
     end if;

  elsif lc_reset_output = ’1’ then

     NEXT_STATE <= RESET_OUTPUT after tpd;

  else

     NEXT_STATE <= SEND_TYPE_TO_FIFO after tpd;

  end if;
```

```
when WAIT_1_2 =>
  ----------------------------------------------
  -- Restliche Signale, fuer Synthese notwendig
  ----------------------------------------------
  --Port-Ausgaenge
   link_status_register_out <= (others => ’0’) after tpd;
   link_status_register_select <= ’0’ after tpd;
   sr_parity_range <= (others => ’0’) after tpd;
   parity_error_select <= ’0’ after tpd;
   ff_token_in <= (others => ’0’) after tpd;
   eop_eom_signal <= ’0’ after tpd;
   eop_eom_select_signal <= ’0’ after tpd;
   ff_delete_all <= ’0’ after tpd;
   ff_next_token <= ’0’ after tpd;
   dom_controller_in <= ’0’ after tpd;
   dom_reset_in <= ’0’ after tpd;
   dom_fct_received <= ’0’ after tpd;

  --Signale in der Komponente
   tc_shift_signal  <= ’0’ after tpd;
   tc_reset_signal <= ’0’ after tpd;
   next_token_type_select_signal <= ’0’ after tpd;
   token_latch_select_signal <= ’0’ after tpd;
   esc_received_latch_select <= ’0’ after tpd;
   esc_received_latch_in <= ’0’ after tpd;
   OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
  ----------------------------------------------

  if lc_reset_link = ’1’ then

     NEXT_STATE <= RESET_LINK after tpd;

  elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

     --localize_error abfragen
     if link_mode_register_in = ’1’ then
          NEXT_STATE <= RESET_LINK after tpd;
     else
          NEXT_STATE <= DISCONNECTION_ERROR after tpd;
     end if;

  elsif lc_reset_output = ’1’ then

     NEXT_STATE <= RESET_OUTPUT after tpd;

  else

     NEXT_STATE <= CHECK_NEXT_TOKEN_TYPE after tpd;

  end if;


when WAIT_1_3 =>
  esc_received_latch_select <= ’1’ after tpd;
  esc_received_latch_in <= ’0’ after tpd;

  ----------------------------------------------
  -- Restliche Signale, fuer Synthese notwendig
  ----------------------------------------------
  --Port-Ausgaenge
   link_status_register_out <= (others => ’0’) after tpd;
   link_status_register_select <= ’0’ after tpd;
   sr_parity_range <= (others => ’0’) after tpd;
   parity_error_select <= ’0’ after tpd;
   ff_token_in <= (others => ’0’) after tpd;
   eop_eom_signal <= ’0’ after tpd;
   eop_eom_select_signal <= ’0’ after tpd;
   ff_delete_all <= ’0’ after tpd;
   ff_next_token <= ’0’ after tpd;
   dom_controller_in <= ’0’ after tpd;
   dom_reset_in <= ’0’ after tpd;
   dom_fct_received <= ’0’ after tpd;

  --Signale in der Komponente
   tc_shift_signal  <= ’0’ after tpd;
   tc_reset_signal <= ’0’ after tpd;
   next_token_type_select_signal <= ’0’ after tpd;
   token_latch_select_signal <= ’0’ after tpd;
   OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
  ----------------------------------------------

  if lc_reset_link = ’1’ then

     NEXT_STATE <= RESET_LINK after tpd;

  elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

     --localize_error abfragen
     if link_mode_register_in = ’1’ then
          NEXT_STATE <= RESET_LINK after tpd;
     else
          NEXT_STATE <= DISCONNECTION_ERROR after tpd;
     end if;

  elsif lc_reset_output = ’1’ then

     NEXT_STATE <= RESET_OUTPUT after tpd;

  else

     NEXT_STATE <= CHECK_NEXT_TOKEN_TYPE after tpd;

  end if;
```

```
when WAIT_6_0 =>

  --------------------------------------------
  -- Restliche Signale, fuer Synthese notwendig
  --------------------------------------------
  --Port-Ausgaenge
   link_status_register_out <= (others => ’0’) after tpd;
   link_status_register_select <= ’0’ after tpd;
   sr_parity_range <= (others => ’0’) after tpd;
   parity_error_select <= ’0’ after tpd;
   ff_token_in <= (others => ’0’) after tpd;
   eop_eom_signal <= ’0’ after tpd;
   eop_eom_select_signal <= ’0’ after tpd;
   ff_delete_all <= ’0’ after tpd;
   ff_next_token <= ’0’ after tpd;
   dom_controller_in <= ’0’ after tpd;
   dom_reset_in <= ’0’ after tpd;
   dom_fct_received <= ’0’ after tpd;

  --Signale in der Komponente
   tc_shift_signal  <= ’0’ after tpd;
   tc_reset_signal <= ’0’ after tpd;
   next_token_type_select_signal <= ’0’ after tpd;
   token_latch_select_signal <= ’0’ after tpd;
   esc_received_latch_select <= ’0’ after tpd;
   esc_received_latch_in <= ’0’ after tpd;
   OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
  --------------------------------------------

  if lc_reset_link = ’1’ then

     NEXT_STATE <= RESET_LINK after tpd;

  elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

     --localize_error abfragen
     if link_mode_register_in = ’1’ then
          NEXT_STATE <= RESET_LINK after tpd;
     else
          NEXT_STATE <= DISCONNECTION_ERROR after tpd;
     end if;

  elsif lc_reset_output = ’1’ then

     NEXT_STATE <= RESET_OUTPUT after tpd;

  else

     NEXT_STATE <= WAIT_6_1 after tpd;

  end if;


when WAIT_6_1 =>
  --------------------------------------------
  -- Restliche Signale, fuer Synthese notwendig
  --------------------------------------------
  --Port-Ausgaenge
   link_status_register_out <= (others => ’0’) after tpd;
   link_status_register_select <= ’0’ after tpd;
   sr_parity_range <= (others => ’0’) after tpd;
   parity_error_select <= ’0’ after tpd;
   ff_token_in <= (others => ’0’) after tpd;
   eop_eom_signal <= ’0’ after tpd;
   eop_eom_select_signal <= ’0’ after tpd;
   ff_delete_all <= ’0’ after tpd;
   ff_next_token <= ’0’ after tpd;
   dom_controller_in <= ’0’ after tpd;
   dom_reset_in <= ’0’ after tpd;
   dom_fct_received <= ’0’ after tpd;

  --Signale in der Komponente
   tc_shift_signal  <= ’0’ after tpd;
   tc_reset_signal <= ’0’ after tpd;
   next_token_type_select_signal <= ’0’ after tpd;
   token_latch_select_signal <= ’0’ after tpd;
   esc_received_latch_select <= ’0’ after tpd;
   esc_received_latch_in <= ’0’ after tpd;
   OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
  --------------------------------------------

  if lc_reset_link = ’1’ then

     NEXT_STATE <= RESET_LINK after tpd;

  elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

     --localize_error abfragen
     if link_mode_register_in = ’1’ then
          NEXT_STATE <= RESET_LINK after tpd;
     else
          NEXT_STATE <= DISCONNECTION_ERROR after tpd;
     end if;

  elsif lc_reset_output = ’1’ then

     NEXT_STATE <= RESET_OUTPUT after tpd;

  else

     NEXT_STATE <= WAIT_6_2 after tpd;

  end if;
```

```
when WAIT_6_2 =>
  --------------------------------------------
  -- Restliche Signale, fuer Synthese notwendig
  --------------------------------------------
  --Port-Ausgaenge
   link_status_register_out <= (others => ’0’) after tpd;
   link_status_register_select <= ’0’ after tpd;
   sr_parity_range <= (others => ’0’) after tpd;
   parity_error_select <= ’0’ after tpd;
   ff_token_in <= (others => ’0’) after tpd;
   eop_eom_signal <= ’0’ after tpd;
   eop_eom_select_signal <= ’0’ after tpd;
   ff_delete_all <= ’0’ after tpd;
   ff_next_token <= ’0’ after tpd;
   dom_controller_in <= ’0’ after tpd;
   dom_reset_in <= ’0’ after tpd;
   dom_fct_received <= ’0’ after tpd;

  --Signale in der Komponente
   tc_shift_signal  <= ’0’ after tpd;
   tc_reset_signal <= ’0’ after tpd;
   next_token_type_select_signal <= ’0’ after tpd;
   token_latch_select_signal <= ’0’ after tpd;
   esc_received_latch_select <= ’0’ after tpd;
   esc_received_latch_in <= ’0’ after tpd;
   OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
  --------------------------------------------

  if lc_reset_link = ’1’ then

     NEXT_STATE <= RESET_LINK after tpd;

  elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

     --localize_error abfragen
     if link_mode_register_in = ’1’ then
         NEXT_STATE <= RESET_LINK after tpd;
     else
         NEXT_STATE <= DISCONNECTION_ERROR after tpd;
     end if;

  elsif lc_reset_output = ’1’ then

     NEXT_STATE <= RESET_OUTPUT after tpd;

  else

     NEXT_STATE <= WAIT_6_3 after tpd;

  end if;


when WAIT_6_3 =>
  --------------------------------------------
  -- Restliche Signale, fuer Synthese notwendig
  --------------------------------------------
  --Port-Ausgaenge
   link_status_register_out <= (others => ’0’) after tpd;
   link_status_register_select <= ’0’ after tpd;
   sr_parity_range <= (others => ’0’) after tpd;
   parity_error_select <= ’0’ after tpd;
   ff_token_in <= (others => ’0’) after tpd;
   eop_eom_signal <= ’0’ after tpd;
   eop_eom_select_signal <= ’0’ after tpd;
   ff_delete_all <= ’0’ after tpd;
   ff_next_token <= ’0’ after tpd;
   dom_controller_in <= ’0’ after tpd;
   dom_reset_in <= ’0’ after tpd;
   dom_fct_received <= ’0’ after tpd;

  --Signale in der Komponente
   tc_shift_signal  <= ’0’ after tpd;
   tc_reset_signal <= ’0’ after tpd;
   next_token_type_select_signal <= ’0’ after tpd;
   token_latch_select_signal <= ’0’ after tpd;
   esc_received_latch_select <= ’0’ after tpd;
   esc_received_latch_in <= ’0’ after tpd;
   OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
  --------------------------------------------

  if lc_reset_link = ’1’ then

     NEXT_STATE <= RESET_LINK after tpd;

  elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

     --localize_error abfragen
     if link_mode_register_in = ’1’ then
         NEXT_STATE <= RESET_LINK after tpd;
     else
         NEXT_STATE <= DISCONNECTION_ERROR after tpd;
     end if;

  elsif lc_reset_output = ’1’ then

     NEXT_STATE <= RESET_OUTPUT after tpd;

  else

     NEXT_STATE <= WAIT_6_4 after tpd;

  end if;
```

```
when WAIT_6_4 =>
  --------------------------------------------
  -- Restliche Signale, fuer Synthese notwendig
  --------------------------------------------
  --Port-Ausgaenge
   link_status_register_out <= (others => ’0’) after tpd;
   link_status_register_select <= ’0’ after tpd;
   sr_parity_range <= (others => ’0’) after tpd;
   parity_error_select <= ’0’ after tpd;
   ff_token_in <= (others => ’0’) after tpd;
   eop_eom_signal <= ’0’ after tpd;
   eop_eom_select_signal <= ’0’ after tpd;
   ff_delete_all <= ’0’ after tpd;
   ff_next_token <= ’0’ after tpd;
   dom_controller_in <= ’0’ after tpd;
   dom_reset_in <= ’0’ after tpd;
   dom_fct_received <= ’0’ after tpd;

  --Signale in der Komponente
   tc_shift_signal  <= ’0’ after tpd;
   tc_reset_signal <= ’0’ after tpd;
   next_token_type_select_signal <= ’0’ after tpd;
   token_latch_select_signal <= ’0’ after tpd;
   esc_received_latch_select <= ’0’ after tpd;
   esc_received_latch_in <= ’0’ after tpd;
   OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
  --------------------------------------------

  if lc_reset_link = ’1’ then

     NEXT_STATE <= RESET_LINK after tpd;

  elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

     --localize_error abfragen
     if link_mode_register_in = ’1’ then
         NEXT_STATE <= RESET_LINK after tpd;
     else
         NEXT_STATE <= DISCONNECTION_ERROR after tpd;
     end if;

  elsif lc_reset_output = ’1’ then

     NEXT_STATE <= RESET_OUTPUT after tpd;

  else

     NEXT_STATE <= WAIT_6_5 after tpd;

  end if;


when WAIT_6_5 =>
  eop_eom_signal <= ’0’ after tpd;
  eop_eom_select_signal <= ’0’ after tpd;
  sr_parity_range <= b“1111_1111“ after tpd;
  parity_error_select <= ’1’ after tpd;

  --------------------------------------------
  -- Restliche Signale, fuer Synthese notwendig
  --------------------------------------------
  --Port-Ausgaenge
   link_status_register_out <= (others => ’0’) after tpd;
   link_status_register_select <= ’0’ after tpd;
   ff_token_in <= (others => ’0’) after tpd;
   ff_delete_all <= ’0’ after tpd;
   ff_next_token <= ’0’ after tpd;
   dom_controller_in <= ’0’ after tpd;
   dom_reset_in <= ’0’ after tpd;
   dom_fct_received <= ’0’ after tpd;

  --Signale in der Komponente
   tc_shift_signal  <= ’0’ after tpd;
   tc_reset_signal <= ’0’ after tpd;
   next_token_type_select_signal <= ’0’ after tpd;
   token_latch_select_signal <= ’0’ after tpd;
   esc_received_latch_select <= ’0’ after tpd;
   esc_received_latch_in <= ’0’ after tpd;
  --------------------------------------------

  OVER_NEXT_STATE <= WAIT_1_1 after tpd;

  if lc_reset_link = ’1’ then

     NEXT_STATE <= RESET_LINK after tpd;

  elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

     --localize_error abfragen
     if link_mode_register_in = ’1’ then
         NEXT_STATE <= RESET_LINK after tpd;
     else
         NEXT_STATE <= DISCONNECTION_ERROR after tpd;
     end if;

  elsif lc_reset_output = ’1’ then

     NEXT_STATE <= RESET_OUTPUT after tpd;

  else

     NEXT_STATE <= GET_TOKENS after tpd;

  end if;
```

```
when GET_TOKENS =>
  next_token_type_select_signal <= ’1’ after tpd;
  token_latch_select_signal <= ’1’ after tpd;

  ---------------------------------------------
  -- Restliche Signale, fuer Synthese notwendig
  ---------------------------------------------
  --Port-Ausgaenge
   link_status_register_out <= (others => ’0’) after tpd;
   link_status_register_select <= ’0’ after tpd;
   sr_parity_range <= (others => ’0’) after tpd;
   parity_error_select <= ’0’ after tpd;
   ff_token_in <= (others => ’0’) after tpd;
   eop_eom_signal <= ’0’ after tpd;
   eop_eom_select_signal <= ’0’ after tpd;
   ff_delete_all <= ’0’ after tpd;
   ff_next_token <= ’0’ after tpd;
   dom_reset_in <= ’0’ after tpd;
   dom_fct_received <= ’0’ after tpd;

  --Signale in der Komponente
   tc_shift_signal  <= ’0’ after tpd;
   esc_received_latch_select <= ’0’ after tpd;
   esc_received_latch_in <= ’0’ after tpd;
   OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
  ---------------------------------------------

  if lc_reset_link = ’1’ then

     dom_controller_in <= ’0’ after tpd;
     tc_reset_signal <= ’0’ after tpd;
     NEXT_STATE <= RESET_LINK after tpd;

  elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

     dom_controller_in <= ’0’ after tpd;
     tc_reset_signal <= ’0’ after tpd;

     --localize_error abfragen
     if link_mode_register_in = ’1’ then
         NEXT_STATE <= RESET_LINK after tpd;
     else
         NEXT_STATE <= DISCONNECTION_ERROR after tpd;
     end if;

  elsif lc_reset_output = ’1’ then

     dom_controller_in <= ’0’ after tpd;
     tc_reset_signal <= ’0’ after tpd;
     NEXT_STATE <= RESET_OUTPUT after tpd;

  elsif sr_parity_error = ’1’ then

     if link_mode_register_in = ’1’ then
         --Gegenteil von elsif
         dom_controller_in <= ’0’ after tpd;
         tc_reset_signal <= ’0’ after tpd;
         NEXT_STATE <= RESET_LINK after tpd;
     else
         --Gegenteil von elsif
         dom_controller_in <= ’0’ after tpd;
         tc_reset_signal <= ’0’ after tpd;
         NEXT_STATE <= PARITY_ERROR after tpd;
     end if;

  elsif tc_full_signal = ’1’ and ff_free = ’0’ then

     dom_controller_in <= ’1’ after tpd;
     tc_reset_signal <= ’1’ after tpd;
     NEXT_STATE <= OVER_STATE after tpd;

  else

     --Gegenteil von elsif
     dom_controller_in <= ’0’ after tpd;
     tc_reset_signal <= ’0’ after tpd;
     NEXT_STATE <= OVER_STATE after tpd;

  end if;


when CHECK_FCT_NUL_OR_EOP_EOM =>
  ---------------------------------------------
  -- Restliche Signale, fuer Synthese notwendig
  ---------------------------------------------
  --Port-Ausgaenge
   link_status_register_out <= (others => ’0’) after tpd;
   link_status_register_select <= ’0’ after tpd;
   sr_parity_range <= (others => ’0’) after tpd;
   parity_error_select <= ’0’ after tpd;
   ff_token_in <= (others => ’0’) after tpd;
   eop_eom_signal <= ’0’ after tpd;
   eop_eom_select_signal <= ’0’ after tpd;
   ff_delete_all <= ’0’ after tpd;
   ff_next_token <= ’0’ after tpd;
   dom_controller_in <= ’0’ after tpd;
   dom_reset_in <= ’0’ after tpd;

  --Signale in der Komponente
   tc_shift_signal  <= ’0’ after tpd;
   tc_reset_signal <= ’0’ after tpd;
   next_token_type_select_signal <= ’0’ after tpd;
   token_latch_select_signal <= ’0’ after tpd;
   OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
  ---------------------------------------------
```

```
  -- wenn FCT nicht in NUL-Token

  if lc_reset_link = ’1’ then

     dom_fct_received <= ’0’ after tpd;
     esc_received_latch_select <= ’0’ after tpd;
     esc_received_latch_in <= ’0’ after tpd;
     NEXT_STATE <= RESET_LINK after tpd;

  elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

     dom_fct_received <= ’0’ after tpd;
     esc_received_latch_select <= ’0’ after tpd;
     esc_received_latch_in <= ’0’ after tpd;

     --localize_error abfragen
     if link_mode_register_in = ’1’ then
         NEXT_STATE <= RESET_LINK after tpd;
     else
         NEXT_STATE <= DISCONNECTION_ERROR after tpd;
     end if;

  elsif lc_reset_output = ’1’ then

     dom_fct_received <= ’0’ after tpd;
     esc_received_latch_select <= ’0’ after tpd;
     esc_received_latch_in <= ’0’ after tpd;
     NEXT_STATE <= RESET_OUTPUT after tpd;

  elsif   token_latch_signal(0 to 1) = b“00“ and
          esc_received_latch_out = ’0’ then

     dom_fct_received <= ’1’ after tpd;
     --Gegenteil von 2.elsif
     esc_received_latch_select <= ’0’ after tpd;
     esc_received_latch_in <= ’0’ after tpd;
     NEXT_STATE <= WAIT_1_2 after tpd;

   --esc_received_latch ruecksetzen
  elsif   token_latch_signal(0 to 1) = b“00“ and
          esc_received_latch_out = ’1’ then

     --Gegenteil von if
     dom_fct_received <= ’0’ after tpd;
     --Gegenteil von 2.elsif
     esc_received_latch_select <= ’0’ after tpd;
     esc_received_latch_in <= ’0’ after tpd;
     NEXT_STATE <= WAIT_1_3 after tpd;

  --ESC-Token
  elsif token_latch_signal(0 to 1) = b“11“ then

     --Gegenteil von if
     dom_fct_received <= ’0’ after tpd;

     esc_received_latch_select <= ’1’ after tpd;
     esc_received_latch_in <= ’1’ after tpd;
     NEXT_STATE <= WAIT_1_2 after tpd;

  else

     --Gegenteil von if
     dom_fct_received <= ’0’ after tpd;
     --Gegenteil von 2.elsif
     esc_received_latch_select <= ’0’ after tpd;
     esc_received_latch_in <= ’0’ after tpd;
     NEXT_STATE <= SEND_TYPE_TO_FIFO after tpd;

  end if;


when SEND_TYPE_TO_FIFO =>
  -- esc_received_latch ruecksetzen
  esc_received_latch_select <= ’1’ after tpd;
  esc_received_latch_in <= ’0’ after tpd;

  ff_token_in(0) <= token_latch_signal(7) after tpd;
  ff_token_in(1) <= token_latch_signal(6) after tpd;
  ff_token_in(2) <= token_latch_signal(5) after tpd;
  ff_token_in(3) <= token_latch_signal(4) after tpd;
  ff_token_in(4) <= token_latch_signal(3) after tpd;
  ff_token_in(5) <= token_latch_signal(2) after tpd;
  ff_token_in(6) <= token_latch_signal(1) after tpd;
  ff_token_in(7) <= token_latch_signal(0) after tpd;

  ff_next_token <= ’1’ after tpd;
  --token_counter
  tc_shift_signal <= ’1’ after tpd;

  --------------------------------------------
  -- Restliche Signale, fuer Synthese notwendig
  --------------------------------------------
  --Port-Ausgaenge
   link_status_register_out <= (others => ’0’) after tpd;
   link_status_register_select <= ’0’ after tpd;
   sr_parity_range <= (others => ’0’) after tpd;
   parity_error_select <= ’0’ after tpd;
   eop_eom_signal <= ’0’ after tpd;
   eop_eom_select_signal <= ’0’ after tpd;
   ff_delete_all <= ’0’ after tpd;
   dom_controller_in <= ’0’ after tpd;
   dom_reset_in <= ’0’ after tpd;
   dom_fct_received <= ’0’ after tpd;
```

```
      --Signale in der Komponente
       tc_reset_signal <= ’0’ after tpd;
       next_token_type_select_signal <= ’0’ after tpd;
       token_latch_select_signal <= ’0’ after tpd;
       OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
      --------------------------------------------

      if lc_reset_link = ’1’ then

         NEXT_STATE <= RESET_LINK after tpd;

      elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

         --localize_error abfragen
         if link_mode_register_in = ’1’ then
              NEXT_STATE <= RESET_LINK after tpd;
         else
              NEXT_STATE <= DISCONNECTION_ERROR after tpd;
         end if;

      elsif lc_reset_output = ’1’ then

         NEXT_STATE <= RESET_OUTPUT after tpd;

      else

         NEXT_STATE <= CHECK_NEXT_TOKEN_TYPE after tpd;

      end if;


    when PARITY_ERROR =>
      link_status_register_select <= ’1’ after tpd;
      link_status_register_out <= (0 => ’1’, 3 => ’1’, others => ’0’)
                                  after tpd;

      --------------------------------------------
      -- Restliche Signale, fuer Synthese notwendig
      --------------------------------------------
      --Port-Ausgaenge
       sr_parity_range <= (others => ’0’) after tpd;
       parity_error_select <= ’0’ after tpd;
       ff_token_in <= (others => ’0’) after tpd;
       eop_eom_signal <= ’0’ after tpd;
       eop_eom_select_signal <= ’0’ after tpd;
       ff_delete_all <= ’0’ after tpd;
       ff_next_token <= ’0’ after tpd;
       dom_controller_in <= ’0’ after tpd;
       dom_reset_in <= ’0’ after tpd;
       dom_fct_received <= ’0’ after tpd;

      --Signale in der Komponente
       tc_shift_signal  <= ’0’ after tpd;
       tc_reset_signal <= ’0’ after tpd;
       next_token_type_select_signal <= ’0’ after tpd;
       token_latch_select_signal <= ’0’ after tpd;
       esc_received_latch_select <= ’0’ after tpd;
       esc_received_latch_in <= ’0’ after tpd;
       OVER_NEXT_STATE <= WAIT_1_1 after tpd; -- dummy value
      --------------------------------------------

      if lc_reset_link = ’1’ then

         NEXT_STATE <= RESET_LINK after tpd;

      elsif dva_data_valid = ’0’ and ls_token_received = ’1’ then

         --localize_error abfragen
         if link_mode_register_in = ’1’ then
              NEXT_STATE <= RESET_LINK after tpd;
         else
              NEXT_STATE <= DISCONNECTION_ERROR after tpd;
         end if;

      elsif lc_reset_output = ’1’ then

         NEXT_STATE <= RESET_OUTPUT after tpd;

      else

         NEXT_STATE <= PARITY_ERROR after tpd;

      end if;

   end case;
  end process;

end behaviour;
--------------------------------------------------------------------
```

```
--**********************************************************************
--**********************************************************************
--**********************************************************************
--**********************************************************************




------------------------------------------------------------------------
-- Mit SYNOPSIS sythetisiertes Gattermodell der Finite State Machine
-- des Controllers.
------------------------------------------------------------------------
library IEEE;

use IEEE.std_logic_1164.all;

library Gates;
use Gates.gates.all;
use work.types.all;

entity controller_fsm is

   port(  clk_phi2, link_mode_register_in,
          lc_reset_link, lc_start_link,
          lc_reset_output, ls_token_received,
          dva_data_valid, sr_parity_error,
          sr_data_out_0, sr_data_out_1,
          sr_control_token_flag,
          ff_free :                          in std_logic;
          link_status_register_out :         out data32;
          link_status_register_select :      out std_logic;
          sr_parity_range :                  out std_logic_vector (0 to 7);
          parity_error_select :              out std_logic;
          ff_token_in :                      out std_logic_vector (0 to 7);
          ff_delete_all, ff_next_token,
          dom_controller_in, dom_reset_in,
          dom_fct_received, tc_shift_signal,
          tc_reset_signal :                  out std_logic;
          tc_full_signal :                   in std_logic;
          token_latch_signal :               in std_logic_vector (0 to 7);
          token_latch_select_signal :        out std_logic;
          next_token_type_signal :           in std_logic;
          next_token_type_select_signal,
          eop_eom_signal,
          eop_eom_select_signal,
          esc_received_latch_select,
          esc_received_latch_in :            out std_logic;
          esc_received_latch_out :           in std_logic);

end controller_fsm;

architecture SYN_behaviour of controller_fsm is

   component and2
      port( i0, i1 : in std_logic;  o : out std_logic);
   end component;

   component nand2
      port( i0, i1 : in std_logic;  o : out std_logic);
   end component;

   component nand3
      port( i0, i1, i2 : in std_logic;  o : out std_logic);
   end component;

   component or2
      port( i0, i1 : in std_logic;  o : out std_logic);
   end component;

   component nand4
      port( i0, i1, i2, i3 : in std_logic;  o : out std_logic);
   end component;

   component nor2
      port( i0, i1 : in std_logic;  o : out std_logic);
   end component;

   component nor3
      port( i0, i1, i2 : in std_logic;  o : out std_logic);
   end component;

   component and3
      port( i0, i1, i2 : in std_logic;  o : out std_logic);
   end component;

   component or3
      port( i0, i1, i2 : in std_logic;  o : out std_logic);
   end component;

   component and4
      port( i0, i1, i2, i3 : in std_logic;  o : out std_logic);
   end component;

   component inv
      port( i : in std_logic;  o : out std_logic);
   end component;

   component or4
      port( i0, i1, i2, i3 : in std_logic;  o : out std_logic);
   end component;

   component mux2
      port( i0, i1, sel : in std_logic;  o : out std_logic);
   end component;
```

```
   component d_ff
      port( d, clk : in std_logic;  q, qn : out std_logic);
   end component;

   signal  Logic0,
           STATE_0_port,
           STATE_1_port,
           STATE_2_port,
           STATE_3_port,
           STATE_4_port,
           NEXT_STATE_0_port,
           NEXT_STATE_1_port,
           NEXT_STATE_2_port,
           NEXT_STATE_3_port,
           NEXT_STATE_4_port,
           OVER_STATE_0_port,
           OVER_STATE_1_port,
           OVER_STATE_2_port,
           OVER_STATE_3_port,
           OVER_STATE_4_port,
           OVER_NEXT_STATE_3_port,
           dom_reset_in_port,
           tc_shift_signal_port,
           sr_parity_range_1_port,
           sr_parity_range_7_port,
           link_status_register_out_0_port,
           link_status_register_out_1_port,
           link_status_register_out_2_port,
           link_status_register_out_3_port,
           link_status_register_out_4_port,
           link_status_register_out_5_port,
           esc_received_latch_in_port,
           token_latch_select_signal_port,
           eop_eom_signal_port,
           ff_delete_all_port,
           n2954, n2955, n2956, n2957, n2958,
           n2959, n2960, n2961, n2962, n2963,
           n2964, n2965, n2966, n2967, n2968,
           n2969, n2970, n2971, n2972, n2973,
           n2974, n2975, n2976, n2977, n2978,
           n2979, n2980, n2981, n2982, n2983,
           n2984, n2985, n2986, n2987, n2988,
           n2989, n2990, n2991, n2992, n2993,
           n2994, n2995, n2996, n2997, n2998,
           n2999, n3000, n3001, n3002, n3003,
           n3004, n3005, n3006, n3007, n3008,
           n3009, n3010, n3011, n3012, n3013,
           n3014, n3015, n3016, n3017, n3018,
           n3019, n3020, n3021, n3022, n3023,
           n3024, n3025, n3026, n3027, n3028,
           n3029, n3030, n3031, n3032, n3033,
           n3034, n3035, n3036, n3037, n3038,
           n3039, n3040, n3041, n3042, n3043,
           n3044, n3045, n3046, n3047, n3048,
           n3049, n3050, n3051, n3052, n3053,
           n3054, n3055, n3056, n3057, n3058,
           n3059, n3060, n3061, n3062, n3063,
           n3064, n3065, n3066, n3067, n3068,
           n3069, n3070, n3071, n3072, n3073,
           n3074, n3075, n3076, n3077, n3078,
           n3079, n3080, n3081, n3082, n3083,
           n3084, n3085, n3086, n3087, n3088 :       std_logic;

begin
   link_status_register_out <= (
          link_status_register_out_0_port,
          link_status_register_out_1_port,
          link_status_register_out_2_port,
          link_status_register_out_3_port,
          link_status_register_out_4_port,
          link_status_register_out_5_port,
          Logic0, Logic0, Logic0, Logic0,
          Logic0, Logic0, Logic0, Logic0,
          Logic0, Logic0, Logic0, Logic0,
          Logic0, Logic0, Logic0, Logic0,
          Logic0, Logic0, Logic0, Logic0,
          Logic0, Logic0, Logic0, Logic0,
          Logic0, Logic0 );

  sr_parity_range <= (
          sr_parity_range_1_port,
          sr_parity_range_1_port,
          sr_parity_range_7_port,
          sr_parity_range_7_port,
          sr_parity_range_7_port,
          sr_parity_range_7_port,
          sr_parity_range_7_port,
          sr_parity_range_7_port );

  ff_delete_all            <=     ff_delete_all_port;
  ff_next_token            <=     tc_shift_signal_port;
  dom_reset_in             <=     dom_reset_in_port;
  tc_shift_signal          <=     tc_shift_signal_port;
  token_latch_select_signal<=     token_latch_select_signal_port;
  eop_eom_signal           <=     eop_eom_signal_port;
  esc_received_latch_in    <=     esc_received_latch_in_port;

  Logic0                   <=     '0';

  U724 : and2  port map( i0 => n2954, i1 => n2955,
                        o  => link_status_register_select);
  U725 : nand2 port map( i0 => n2956, i1 => n2957,
                        o  => parity_error_select);
  U726 : nand2 port map( i0 => n2958, i1 => n2959, o => tc_reset_signal);
  U727 : nand3 port map( i0 => n2959, i1 => n2960,
                        i2 => n2961, o => dom_controller_in);
```

```
U728 : or2   port map( i0 => link_status_register_out_2_port,
                       i1 => ff_delete_all_port, o => dom_reset_in_port);
U729 : nand3 port map( i0 => n2962, i1 => n2963,
                       i2 => next_token_type_signal,
                       o  => OVER_NEXT_STATE_3_port);
U730 : nand2 port map( i0 => n2958, i1 => n2964,
                       o  => eop_eom_select_signal);
U731 : nand2 port map( i0 => n2965, i1 => n2966,
                       o  => next_token_type_select_signal);
U732 : nand2 port map( i0 => n2965, i1 => n2967, o => dom_fct_received);
U733 : nand4 port map( i0 => n2968, i1 => n2969, i2 => n2970,
                       i3 => n2958, o => esc_received_latch_select);
U734 : or2   port map( i0 => link_status_register_out_4_port,
                       i1 => link_status_register_out_3_port,
                       o  => link_status_register_out_0_port);
U735 : or2   port map( i0 => link_status_register_out_5_port,
                       i1 => n2971, o=>ink_status_register_out_1_port);
U736 : or2   port map( i0 => sr_parity_range_7_port,
                       i1 => eop_eom_signal_port,
                       o  => sr_parity_range_1_port);
U737 : and2  port map( i0 => token_latch_signal(7),
                       i1 => tc_shift_signal_port,
                       o  => ff_token_in(0));
U738 : and2  port map( i0 => token_latch_signal(6),
                       i1 => tc_shift_signal_port,
                       o  =>  ff_token_in(1));
U739 : and2  port map( i0 => token_latch_signal(5),
                       i1 => tc_shift_signal_port,
                       o  => ff_token_in(2));
U740 : and2  port map( i0 => token_latch_signal(4),
                       i1 => tc_shift_signal_port,
                       o  => ff_token_in(3));
U741 : and2  port map( i0 => token_latch_signal(3),
                       i1 => tc_shift_signal_port,
                       o  => ff_token_in(4));
U742 : and2  port map( i0 => token_latch_signal(2),
                       i1 => tc_shift_signal_port,
                       o  => ff_token_in(5));
U743 : and2  port map( i0 => token_latch_signal(1),
                       i1 => tc_shift_signal_port,
                       o  => ff_token_in(6));
U744 : and2  port map( i0 => token_latch_signal(0),
                       i1 => tc_shift_signal_port,
                       o  => ff_token_in(7));
U745 : nand4 port map( i0 => n2972, i1 => n2973, i2 => n2974,
                       i3 => n2975, o => NEXT_STATE_4_port);
U746 : nand4 port map( i0 => n2976, i1 => n2977, i2 => n2978,
                       i3 => n2979, o => NEXT_STATE_3_port);
U747 : nand4 port map( i0 => n2974, i1 => n2980, i2 => n2981,
                       i3 => n2982, o => NEXT_STATE_2_port);
U748 : nand4 port map( i0 => n2983, i1 => n2984, i2 => n2981,
                       i3 => n2985, o => NEXT_STATE_1_port);
U749 : nand4 port map( i0 => n2976, i1 => n2986, i2 => n2987,
                       i3 => n2988, o => NEXT_STATE_0_port);
U750 : nor2  port map( i0 => n2989, i1 => STATE_3_port, o => n2963);
U751 : and2  port map( i0 => n2991, i1 => n2992, o => n2990);
U752 : nor2  port map( i0 => n2994, i1 => STATE_0_port, o => n2993);
U753 : or2   port map( i0 => STATE_3_port, i1 => STATE_1_port,
                       o  => n2995);
U754 : nor3  port map( i0 => lc_reset_link, i1 => n2997,
                       i2 => lc_reset_output, o => n2996);
U755 : and2  port map( i0 => n2996, i1 => STATE_4_port, o => n2998);
U756 : nand2 port map( i0 => n3000, i1 => n2993, o => n2999);
U757 : and2  port map( i0 => STATE_0_port, i1 => n2994, o => n3001);
U758 : and2  port map( i0 => n3001, i1 => n2955, o => n3002);
U759 : and2  port map( i0 => STATE_0_port, i1 => STATE_2_port, o => n3003);
U760 : and2  port map( i0 => n3003, i1 => n2996, o => n2962);
U761 : and3  port map( i0 => STATE_1_port, i1 => STATE_4_port,
                       i2 => STATE_0_port, o => sr_parity_range_7_port);
U762 : nand2 port map( i0 => n3005, i1 => sr_control_token_flag,
                       o  => n3004);
U763 : nand4 port map( i0 => n3007, i1 => STATE_2_port, i2 => n3008,
                       i3 => n3009, o => n3006);
U764 : and2  port map( i0 => STATE_3_port, i1 => n2989, o => n3010);
U765 : nor2  port map( i0 => STATE_0_port, i1 => STATE_2_port, o => n3011);
U766 : nor3  port map( i0 => n2955, i1 => n2994, i2 => n2989,
                       o  => tc_shift_signal_port);
U767 : nor2  port map( i0 => STATE_0_port, i1 => STATE_3_port, o => n3012);
U768 : nand2 port map( i0 => n2962, i1 => STATE_4_port, o => n3013);
U769 : and2  port map( i0 => token_latch_signal(1),
                       i1 => token_latch_signal(0), o => n3014);
U770 : or2   port map( i0 => n3013, i1 => n3016, o => n3015);
U771 : and2  port map( i0 => n3017, i1 => n3014,
                       o  => esc_received_latch_in_port);
U772 : nor3  port map( i0 => esc_received_latch_out,
                       i1 => token_latch_signal(0),
                       i2 => token_latch_signal(1), o => n3016);
U773 : or3   port map( i0 => STATE_3_port, i1 => STATE_2_port,
                       i2 => n3018, o => n2956);
U774 : and2  port map( i0 => STATE_3_port, i1 => STATE_1_port, o => n3019);
U775 : and3  port map( i0 => n2993, i1 => n2989, i2 => STATE_4_port,
                       o  => token_latch_select_signal_port);
U776 : and3  port map( i0 => n2996, i1 => n3021,
                       i2 => token_latch_select_signal_port,o => n3020);
U777 : and2  port map( i0 => n3022, i1 => n3023, o => n2972);
U778 : and2  port map( i0 => n2957, i1 => n3024, o => n2974);
U779 : and2  port map( i0 => n3010, i1 => n3001,
                       o  => link_status_register_out_5_port);
U780 : nor2  port map( i0 => token_latch_signal(0),
                       i1 => token_latch_signal(1), o => n3025);
U781 : and2  port map( i0 => n2996, i1 => n2955, o => n3026);
U782 : and2  port map( i0 => n3011, i1 => n3010,
                       o  => link_status_register_out_3_port);
U783 : nand2 port map( i0 => n3012, i1 => tc_shift_signal_port,
                       o  => n2970);
```

```
U784 : and2  port map( i0 => n3028, i1 => n3029, o => n3027);
U785 : and3  port map( i0 => n3027, i1 => n2972, i2 => n3030, o => n2981);
U786 : or2   port map( i0 => n3012, i1 => n3000, o => n3031);
U787 : or2   port map( i0 => n3033, i1 => dva_data_valid, o => n3032);
U788 : nor2  port map( i0 => ff_free, i1 => sr_parity_error, o => n3034);
U789 : nor2  port map( i0 => sr_data_out_1, i1 => sr_data_out_0,
                       o  => n3005);
U790 : nor3  port map( i0 => STATE_4_port, i1 => n2997,
                       i2 => lc_reset_output, o => n3009);
U791 : and3  port map( i0 => n3035, i1 => n2986, i2 => n3036, o => n2975);
U792 : and3  port map( i0 => n3037, i1 => n3038, i2 => n3039, o => n2978);
U793 : and4  port map( i0 => n3040, i1 => n3041, i2 => n3042,
                       i3 => n2965, o => n2979);
U794 : and3  port map( i0 => n3037, i1 => n2960, i2 => n3039, o => n3030);
U795 : and4  port map( i0 => n3043, i1 => n2976, i2 => n2977,
                       i3 => n3044, o => n2982);
U796 : and4  port map( i0 => n3045, i1 => n3035, i2 => n3038,
                       i3 => n3046, o => n2985);
U797 : and3  port map( i0 => n3044, i1 => n3046, i2 => n3039, o => n2987);
U798 : and2  port map( i0 => n3027, i1 => n3048, o => n3047);
U799 : and4  port map( i0 => n3049, i1 => n3006, i2 => n3050,
                       i3 => n3047, o => n2988);
U800 : and3  port map( i0 => n3000, i1 => n2955, i2 => n3011,
                       o  => link_status_register_out_4_port);
U801 : and3  port map( i0 => n2963, i1 => n2955, i2 => n3011,
                       o  => link_status_register_out_2_port);
U802 : nand2 port map( i0 => n2996, i1 => n3051, o => n3046);
U803 : nand3 port map( i0 => n2993, i1 => n3052, i2 => n3026, o => n3044);
U804 : nand4 port map( i0 => n2962, i1 => n3000, i2 => n2955,
                       i3 => n3004, o => n2960);
U805 : nand2 port map( i0 => link_status_register_out_5_port,
                       i1 => n2996, o => n3038);
U806 : nand3 port map( i0 => n2963, i1 => n3053, i2 => n2962, o => n3037);
U807 : nand3 port map( i0 => n2996, i1 => n2993, i2 => n3019, o => n3039);
U808 : nand2 port map( i0 => n3011, i1 => n2998, o => n2986);
U809 : nand3 port map( i0 => n2998, i1 => n2989, i2 => n3001, o => n3035);
U810 : nand3 port map( i0 => n2998, i1 => STATE_1_port, i2 => n3011,
                       o  => n3045);
U811 : nand2 port map( i0 => n3054, i1 => n3055, o => n2977);
U812 : nand2 port map( i0 => n3025, i1 => n3017, o => n2976);
U813 : nand2 port map( i0 => n3002, i1 => n3000, o => n2958);
U814 : or2   port map( i0 => n3006, i1 => lc_reset_link, o => n2965);
U815 : nand4 port map( i0 => tc_full_signal, i1 => n3056,
                       i2 => n3034, i3 => n2998, o => n2959);
U816 : inv   port map( i  => OVER_NEXT_STATE_3_port,
                       o  => eop_eom_signal_port);
U817 : nand2 port map( i0 => n3056, i1 => STATE_4_port, o => n2966);
U818 : or2   port map( i0 => n2989, i1 => n3012, o => n3057);
U819 : nand2 port map( i0 => n3057, i1 => n2994, o => n3058);
U820 : nand2 port map( i0 => n2999, i1 => n3058, o => n2954);
U821 : and2  port map( i0 => n3056, i1 => n2955, o => n2971);
U822 : nand3 port map( i0 => n3003, i1 => n2955, i2 => n3010, o => n2969);
U823 : nand2 port map( i0 => esc_received_latch_in_port, i1 => n3000,
                       o  => n2968);
U824 : nand3 port map( i0 => n2963, i1 => n2955, i2 => n2962, o => n2964);
U825 : nand3 port map( i0 => n3054, i1 => n3000, i2 => n3016, o => n2967);
U826 : or3   port map( i0 => STATE_3_port, i1 => n3007, i2 => n3060,
                       o  => n3059);
U827 : nand2 port map( i0 => n2995, i1 => n3059, o => n3061);
U828 : nand3 port map( i0 => n3061, i1 => n2955, i2 => n2993, o => n2961);
U829 : or2   port map( i0 => n2956, i1 => n3060, o => n3024);
U830 : nand2 port map( i0 => eop_eom_signal_port, i1 => n2955, o => n2957);
U831 : or4   port map( i0 => n3015, i1 => n2995, i2 => n3014, i3 => n3025,
                       o  => n3023);
U832 : nand3 port map( i0 => n3002, i1 => n2996, i2 => n3019, o => n3022);
U833 : nand2 port map( i0 => n3019, i1 => n2962, o => n3036);
U834 : nand2 port map( i0 => n3020, i1 => OVER_STATE_4_port, o => n2973);
U835 : or2   port map( i0 => n3016, i1 => n3014, o => n3055);
U836 : or4   port map( i0 => n3021, i1 => n2999, i2 => n3062,
                       i3 => link_mode_register_in, o => n3042);
U837 : nand2 port map( i0 => link_status_register_out_3_port,
                       i1 => n3026, o => n3041);
U838 : nand2 port map( i0 => OVER_STATE_3_port, i1 => n3020, o => n3040);
U839 : nand3 port map( i0 => n3026, i1 => STATE_3_port, i2 => n3063,
                       o  => n3029);
U840 : or2   port map( i0 => n2970, i1 => n3060, o => n3028);
U841 : nand4 port map( i0 => lc_start_link, i1 => n3002, i2 => n2996,
                       i3 => n2963, o => n3043);
U842 : nand2 port map( i0 => OVER_STATE_2_port, i1 => n3020, o => n2980);
U843 : and3  port map( i0 => n2963, i1 => n3065, i2 => n3002, o => n3064);
U844 : or3   port map( i0 => dom_reset_in_port,
                       i1 => link_status_register_out_4_port,
                       i2 => n3064, o => n3051);
U845 : or2   port map( i0 => n3031, i1 => n3067, o => n3066);
U846 : nand2 port map( i0 => n3066, i1 => n3032, o => n3068);
U847 : nand2 port map( i0 => n3032, i1 => n2955, o => n3069);
U848 : nand2 port map( i0 => n3069, i1 => n3068, o => n3070);
U849 : nand3 port map( i0 => n3070, i1 => n3071, i2 => lc_reset_output,
                       o  => n2984);
U850 : nand2 port map( i0 => OVER_STATE_1_port, i1 => n3020, o => n2983);
U851 : nor2  port map( i0 => STATE_2_port, i1 => STATE_3_port, o => n3072);
U852 : or4   port map( i0 => n2955, i1 => n3072, i2 => n3012, i3 => n3000,
                       o  => n3073);
U853 : nand3 port map( i0 => n3074, i1 => n3075, i2 => n2999, o => n2992);
U854 : or2   port map( i0 => n3021, i1 => lc_reset_output, o => n3076);
U855 : nand2 port map( i0 => n3076, i1 => n3032, o => n2991);
U856 : nand2 port map( i0 => n3077, i1 => link_mode_register_in,
                       o  => n3049);
U857 : nand2 port map( i0 => lc_reset_link, i1 => n3073, o => n3050);
U858 : nand2 port map( i0 => OVER_STATE_0_port, i1 => n3020, o => n3048);
U859 : nand3 port map( i0 => n2963, i1 => n2994, i2 => n2997, o => n3075);
U860 : nand2 port map( i0 => n3007, i1 => n2999, o => n3052);
U861 : nand2 port map( i0 => n2997, i1 => n3031, o => n3074);
U862 : mux2  port map( i0 => n2963, i1 => n3000, sel => STATE_0_port,
                       o  =>  n3008);
```

```
U863 : mux2  port map( i0 => STATE_2_port, i1 => n3011,
                       sel => STATE_1_port, o => n3063);
U864 : mux2  port map( i0 => n2997, i1 => n2990, sel => STATE_4_port,
                       o  => n3077);
U865 : inv port map( i => ls_token_received, o => n3033);
U866 : inv port map( i => STATE_4_port, o => n2955);
U867 : inv port map( i => STATE_2_port, o => n2994);
U868 : inv port map( i => sr_parity_range_7_port, o => n3018);
U869 : inv port map( i => STATE_1_port, o => n2989);
U870 : inv port map( i => lc_reset_link, o => n3071);
U871 : inv port map( i => next_token_type_signal, o => n3053);
U872 : inv port map( i => sr_parity_error, o => n3021);
U873 : inv port map( i => n2998, o => n3062);
U874 : inv port map( i => lc_start_link, o => n3065);
U875 : inv port map( i => n2995, o => n3000);
U876 : inv port map( i => n2996, o => n3060);
U877 : inv port map( i => n2999, o => n3056);
U878 : inv port map( i => n3004, o => n3007);
U879 : inv port map( i => n3013, o => n3054);
U880 : inv port map( i => n3015, o => n3017);
U881 : inv port map( i => n2956, o => n3067);
U882 : inv port map( i => n3032, o => n2997);
U883 : inv port map( i => n2958, o => ff_delete_all_port);
n3078 <= '1';

OVER_STATE_reg_0_label :   d_ff port map( d   => n3078,
                                          clk => clk_phi2,
                                          q   => OVER_STATE_0_port,
                                          qn  => n3079);
STATE_reg_0_label :        d_ff port map( d   => NEXT_STATE_0_port,
                                          clk => clk_phi2,
                                          q   => STATE_0_port,
                                          qn  => n3080);
STATE_reg_1_label :        d_ff port map( d   => NEXT_STATE_1_port,
                                          clk => clk_phi2,
                                         q   => STATE_1_port,
                                          qn  => n3081);
STATE_reg_2_label :        d_ff port map( d   => NEXT_STATE_2_port,
                                          clk => clk_phi2,
                                         q   => STATE_2_port,
                                          qn  => n3082);
STATE_reg_3_label :        d_ff port map( d   => NEXT_STATE_3_port,
                                          clk => clk_phi2,
                                          q   => STATE_3_port,
                                          qn  => n3083);
STATE_reg_4_label :        d_ff port map( d   => NEXT_STATE_4_port,
                                          clk => clk_phi2,
                                          q   => STATE_4_port,
                                          qn  => n3084);
OVER_STATE_reg_1_label :   d_ff port map( d   => OVER_NEXT_STATE_3_port,
                                          clk => clk_phi2,
                                          q   => OVER_STATE_1_port,
                                          qn  => n3085);
OVER_STATE_reg_2_label :   d_ff port map( d   => eop_eom_signal_port,
                                          clk => clk_phi2,
                                          q   => OVER_STATE_2_port,
                                          qn  => n3086);
OVER_STATE_reg_3_label :   d_ff port map( d   => OVER_NEXT_STATE_3_port,
                                          clk => clk_phi2,
                                          q   => OVER_STATE_3_port,
                                          qn  => n3087);
OVER_STATE_reg_4_label :   d_ff port map( d   => eop_eom_signal_port,
                                          clk => clk_phi2,
                                          q   => OVER_STATE_4_port,
                                          qn  => n3088);

end SYN_behaviour;
```

```
--***********************************************************************
--***********************************************************************
--***********************************************************************
--***********************************************************************


---------------------------------------------------------------------
-- token_counter
-- zaehlt von 1 bis 8
-- wird gebraucht, um nach 8 empfangenen tokens ein FCT-token
-- zurueckzuschicken
---------------------------------------------------------------------
library  IEEE;
use      IEEE.std_logic_1164.all;
library  Gates;
use      Gates.gates.all;

---------------------------------------------------------------------
entity token_counter is
  port (   gnd:       in  std_logic;
           tc_shift:  in  std_logic;
           tc_reset:  in  std_logic;
           tc_full:   out std_logic
  );
end token_counter;

---------------------------------------------------------------------

architecture structure of token_counter is

  signal a0, a1, a2, a3, a4, a5, a6:               std_logic;
  signal d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7:   std_logic;
  signal f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7:   std_logic;
  signal nr :                                      std_logic;
  signal cl1 :                                     std_logic;
  signal cl2 :                                     std_logic;


begin

  i_0: inv   port map (i => tc_reset, o => nr);

  o_0: nor2  port map (i0 => tc_shift, i1 => tc_reset, o => cl1);
  i_1: inv   port map (i  => cl1, o => cl2);

  a_0: and2  port map (i0 => nr, i1 => tc_shift, o => d_0);
  l_0: d_ff  port map (clk => cl2, d => d_0, q => a0, qn => f_0);

  a_1: and2  port map (i0 => nr, i1 => a0, o => d_1);
  l_1: d_ff  port map (clk => cl2, d => d_1, q => a1, qn => f_1);

  a_2: and2  port map (i0 => nr, i1 => a1, o => d_2);
  l_2: d_ff  port map (clk => cl2, d => d_2, q => a2, qn => f_2);

  a_3: and2  port map (i0 => nr, i1 => a2, o => d_3);
  l_3: d_ff  port map (clk => cl2, d => d_3, q => a3, qn => f_3);

  a_4: and2  port map (i0 => nr, i1 => a3, o => d_4);
  l_4: d_ff  port map (clk => cl2, d => d_4, q => a4, qn => f_4);

  a_5: and2  port map (i0 => nr, i1 => a4, o => d_5);
  l_5: d_ff  port map (clk => cl2, d => d_5, q => a5, qn => f_5);

  a_6: and2  port map (i0 => nr, i1 => a5, o => d_6);
  l_6: d_ff  port map (clk => cl2, d => d_6, q => a6, qn => f_6);

  a_7: and2  port map (i0 => nr, i1 => a6, o => d_7);
  l_7: d_ff  port map (clk => cl2, d => d_7, q => tc_full, qn => f_7);


end structure;
---------------------------------------------------------------------
```

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[ArAA 90] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, D. Powell: "Fault injection for dependability validation: a methodology and some applications". *IEEE Transactions on Software Engineering*, Vol. 16, No. 2, February 1990, pp. 166-182

[Ash 90] P. Ashenden: "The VHDL-cookbook", Technical Report, University of Adelaide, Adelaide, Australien, 1990.

[Bal 96] F. Balbach: "VHDL-basierte Fehleranalyse eines ATM-Switches", In: *Simulationsbasierte Zuverlässigkeitsanalyse*, *Interner Report Nr. 6/96*, Universität Erlangen-Nürnberg, IMMD III, Erlangen, 1996.

[BaCS 90] J. Barton, E. Czeck, Z. Segall, D. Siewiorek: "Fault Injection Experiments using FIAT". *IEEE Transaction on Computers,* Vol. 39, No. 4, April 1990, S. 575-582.

[BhMc 83] T.N. Bhar, E.J. McMahon: "Electronic Discharge Control, Successful Methods for Microelectronics Design and Manufacturing", Hayden Book Company, Rochelle Park, New Jersey, 1983.

[Ble 96] A. Bleck: "Praktikum des modernen VLSI-Entwurfs", Teubner Verlag, Stuttgart, 1996.

[Bog 96] R. Bogendörfer: "VHDL-basierte Fehleranalyse von Netzwerkelementen“, Diplomarbeit am IMMD3 der Universität Erlangen-Nürnberg, Erlangen, 1996.

[Bra 68] P. T. Brady : „A Statistical Analysis of On-Off Patterns in 16 Conversations“, Bell System Tech. J., Vol 47, Nr. 1, S. 73-91, 1968.

[ChIC 90] G. Choi, R. Iyer, V. Carreno: "Simulated fault injection: A methodology to evaluate fault tolerant microprocessor architectures". *IEEE Trans. Reliability*, Vol. 39, No. 4, Oktober 1990, S. 486-490.

[CaMS 95] J. Carreira, H. Madeira, J. G. Silva: "Xception: Software Fault Injection and Monitoring in Processor Functional Units" *Preprints of the DCCA-5, Working Conference on Dependable Computing for Critical Applications*, Urbana Champaign , USA, Beckman Institute, September 27-29, 1995, S. 135-149.

[ChRC 93] H. Cha, E. Rudnick, G. Choi, J. Patel, R. Iyer: "A Fast and Accurate Gate-Level Transient Fault Simulation Environment", *Proc. 23rd Symp. on Fault-Tolerant Computing (FTCS-23)*, Toulouse, Frankreich, 1993, S. 310-319.

[ClPr 94] J. A. Clark, D. K. Pradhan: "REACT: An Integrated Tool for the Design of Dependable Computer Systems", In *Foundations of Dependable Computing, Models and Frameworks for Dependable Systems*, G. M. Koob, C. G. Lau (ed.), Kluwer, 1994, S. 169-192.

[CzSi 90] E. Czeck, D. Siewiorek: "Effects of Transient Gate-Level Faults on Program Behavior", *Proc. 20th Symp. on Fault Tolerant Computing (FTCS-20)*, Newcastle Upon Tyne, Juni 1990, S. 236-243.

[EcLe 92] K. Echtle, M. Leu: "The EFA Fault Injector for Fault Tolerant Distributred System Testing", *Proc. IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, Amherst (MA), USA, 1992, S. 28-35.

[Fle 97] Fleischmann, M: "ATM-Lastmodellierung zur Fehleranalyse", Studienarbeit am IMMD3 der Universität Erlangen-Nürnberg, Erlangen, 1997.

[GoIy 90] K. K. Goswami, R. K. Iyer: "DEPEND: A Design Environment for Prediction and Evaluation of System Dependability", *Proc. 9th Digital Avionics Systems Conference*, Oktober 1990.

[GuKT 89] U. Gunneflo, J. Karlsson, J. Torin: "Evaluation of error detection schemes using fault injection by heavy-ion radiation." *Proc. 19th Symp. on Fault-Tolerant Computing (FTCS-19)*, Chicago, Illinois, 21-23 Juni 1989, S.340-347.

[GuSi 95] J. Güthoff, V. Sieh: "Combining Software-Implemented and Simulation-Based Fault Injection into a Single Fault Injection Method", *Proc. 25th Symp. on Fault-Tolerant Computing (FTCS-25)*, Pasadena, Kalifornien, Juni 1995, S. 196-206.

[HaRo 93] S. Han, H. A. Rosenberg, K. G. Shin: "*DOCTOR: An IntegrateD SOftware Fault InjeCTiOn EnviRonment*", Technical Report Univ. of Michigan, Dezember 1993.

[HeGo 95] A. Hein, K. K. Goswami: "Combined Performance and Dependability Evaluation with Conjoint Simulation", *Proc. of 7th European Simulation Symposium*, Erlangen-Nürnberg, 26-28 Oktober, 1995, S. 365-369.

[INM 93a] Inmos: "The T9000 Instruction Set Manual", SGS-Thomson Microelectronics, 1st edition, 1993.

[INM 93b] Inmos: "The T9000 Transputer Hardware Reference Manual", SGS-Thomson Microelectronics, 1st edition, 1993.

[JeAR 94] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, J. Karlsson: "Fault Injection into VHDL Models: The MEFISTO Tool". *Proc. 24th Symp. on Fault Tolerant Computing, (FTCS-24)*, IEEE, Austin, Texas, USA, 1994, S. 66-75.

[IyRo 86] R.H. Iyer, D. Rosetti: "A measurement based model for workload-dependance of CPU-errors", *IEEE Transactions on Computers*, vol C-35, 1986 Juni, S. 511-519.

[KaKA 92] G. A. Kanawati, N. A. Kanawati, J. A. Abraham: "FERRARI: A Tool for the Validation of System Dependability Properties", *Proc. 22th Symp. on Fault-Tolerant Computing (FTCS-22)*, Boston, Massachusetts, 8-10 Juli, 1992, S. 336-344.

[KaIT 93] W. Kao, R. K. Iyer, D. Tang: "FINE: A Fault Injection and Monitor Environment for Tracing the UNIX System Behavior under Faults", *IEEE Trans. on Soft. Eng.*, Vol. 19, No. 11, Nov. 1993, S. 1105-1118.

[KaFA 95] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, J. Reisinger: "Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture", *Preprints of Fifth International Working Conference on Dependable Computing for Critical Applications (DCCA-5)*, Urbana-Champaign, Illinois, USA, September 27-29, 1995, S. 150-161.

[KaGT 91] J. Karlsson, U. Gunneflo, J. Torin: "Use of Heavy-Ion Radiation from Californium-252 for Fault Injection Experiments" in *Dependable Computing for Critical Applications*, A. Avizienis, J.-C. Laprie (eds.), in series "Dependable Computing and Fault-Tolerant Systems", Vol. 4, Springer-Verlag Wien-New York, 1991, S. 197-212.

[KuKA 94] S. Kumar, R. H. Klenke, J. H. Aylor, B. W. Johnson, R. D. Williams, R. Waxman, "ADEPT: A Unified System Level Modeling Design Environment", *Proceedings of the 1st Annual RASSP Conference*, Arlington, Virginia, 15-18. August, 1994, S. 114-123.

[MaRS 94] H. Madeira, M. Rela, J. G. Silva: "RIFLE: A General Purpose Pin-Level Fault Injector", *Proc. First European Dependable Computing Conference (EDCC-1)*, Berlin, Germany, Springer Verlag, 4.-6. Oktober, 1994, S. 199-216.

[Mard 86] M. Mardiguian: "Electrostatic Discharge, Understand, Simulate and Fix ESD Problems", Interference Control Technologies, Inc. Gainesville, Virginia 1986.

[MaTW 93] M.D. May, P.W. Thomson, P.H. Welch: "Networks, Routers & Transputers - Function, Performance and Application", IOS-Press, Amsterdam, 1993.

[PiDu 81] D.G. Pierce, D.L. Durgin: "An Overview of Electrical Overstress Effects on Semiconductor Devices", In *Proceedings of the 3rd Annual EOS/ESD Symposium*, 1981.

[Pry 93] M. de Prycker: "Asynchronous Transfer Mode - Solution for Broadband ISDN", Ellis Horwood, New York, 1993.

[RiOK 93] M. Rimén, J. Ohlsson, J. Karlsson, E. Jenn, J. Arlat: "Design Guidelines of a VHDL-based Simulation Tool for the Validation of Fault Tolerance", *Proc. 1st ESPRIT Basic Research Project PDCS-2 Open Workshop*, LAAS-CNRS, Toulouse, Frankreich, September 1993, S. 461-483.

[RiOT 94] M. Rimén, J. Ohlsson, J. Torin: "On Microprocessor Error Behavior Modeling". *Proc. IEEE 24th International Symposium on Fault-Tolerant Computing (FTCS-24)*, Austin, Texas, USA, 1994, S. 76-85.

[SeVS 88] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Rancey, A. Robinson, T. Lin: "FIAT — Fault Injection Based Automated Testing Environment", *Proc. 18th Symp.on Fault-Tolerant Computing (FTCS-18)*, Tokyo, Japan, IEEE CS Press, Juni 1988, S. 102-107.

[SchA 92] J. M. Schoen, S. J. Adamus: "Performance and Fault Modeling in VHDL", Prentice Hall, Englewood Cliffs, 1992.

[SGS 95] SGS-Thomson Microelectronics: "STC104 engineering datasheet", Document Number: 42 1470 06, SGS-Thomson, 1995.

[ShMF 85] J.P. Shen, W. Maly, F.J. Ferguson: "Inductive Fault Analysis of MOS Integrated Curcuits", IEEE Design and Test of Computers, 2, Nr.4, Dezember 1985.

[Sie 98] V. Sieh : „Effiziente Erstellung und Auswertung von Rechnermodellen zur Detaillierten Zuverlässigkeitsanalyse“, Dissertation, Universität Erlangen-Nürnberg, 1998.

[SiBT 97] V. Sieh, F. Balbach, O. Tschäche, "VERIFY: Zuverlässigkeitsanalyse unter Verwendung von VHDL-Modellen mit integrierter Fehlerbeschreibung", *Tagungsband 9. Workshop „Testmethoden und Zuverlässigkeit von Schaltungen und Systemen“*, Universität Bremen, Bremen, Deutschland, 9.-11. März 1997, S. 39-42.

[SiPS 94] V. Sieh, A. Pataricza, B. Sallay, W. Hohl, J. Hönig, B. Benyó, "Fault Injection Based Validation of Fault-Tolerant Multiprocessors", *Proc. µP’94, 8th Symposium on Microcomputer and Microprocessor Applications*, TU Budapest, 1994, S. 85-94.

[SiTB 97a] V. Sieh, O. Tschäche, F. Balbach, "Comparing Different Fault Models Using VERIFY", *Proceedings 6th Conference on Dependable Computing for Critical Applications (DCCA-6)*, Grainau, Deutschland, 5.-7. März 1997, S. 59-76.

[SiTB 97b] V. Sieh, O. Tschäche, F. Balbach, "System Dependability Analysis using VHDL Models with Integrated Fault Descriptions", *Extended Abstracts 8th European Workshop on Dependable Computing (EWDC-8)*, Göteborg, Schweden, 1.-4. April 1997.

[SiTB 97c] V. Sieh, O. Tschäche, F. Balbach, "VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions", *Proceedings 27th Symposium on Fault Tolerant Computing (FTCS-27)*, Seattle (WA), USA, 25.-27. Juni 1997, S. 32-36.

[SiSw 82] D. Siewiorek, R. Swarz: "The Theory and Practice of Reliable Systems Design", Digital Equipment Corporation, 1982.

[Stei 97] A. Steininger, C. Scherrer, "On Finding an Optimal Combination of Error Detection Mechanisms Based on Results of Fault Injection Experiments", *Proceedings 27th Symposium on Fault Tolerant Computing (FTCS-27)*, Seattle (WA), USA, Juni 1997, S.238-247.

[Sti 97] Stiborsky, J.: "Modellierung eines STC104 in VHDL", Diplomarbeit am IMMD3 an der Universität Erlangen-Nürnberg, Erlangen, Juni 1997.

[Swe91] Sweeney, P.: "Codierung zur Fehlererkennung und Fehlerkorrektur", Hanser & Prentice Hall, Englewood Cliffs, 1991.

[YeHA87] Yu-Shuan Yeh, M. Hluchyj, A. S. Acampora: "The Knockout Switch: A Simple, Modular Architecture for High-Performance Packet Switching", IEEE Journal on Selected Areas in Communications, Band 5, Nr. 8, Phoenix, 1987.

Standardisierungsdokumente

[IEEE 93] "*IEEE Standard VHDL Language Reference Manual*", ANSI/IEEE Std 1076-1993, IEEE Inc., 1993.

[G.703] ITU-T Recommendation G.703: "Physical / Electrical Characteristics of Hierarchical Digital Interfaces", International Telecommunication Union, Telecommunication Standardization Sector, 1991.

[G.708] ITU-T Recommendation G.708: "Network Node Interface for the Synchronous DIgital Hierarchy", International Telecommunication Union, Telecommunication Standardization Sector, 1991.

[G.726] ITU Recommendation G.726: "40 , 32, 24, 16 kbit/s Adaptive Differential Pulse Code Modulation (ADPCM)", International Telecommunication Union, Telecommunication Standardization Sector, 1995.

[I.311] ITU-T Recommendation I.311: "B-ISDN General Network Aspects", International Telecommunication Union, Telecommunication Standardization Sector, 1993.

[I.321] ITU-T Recommendation I.321: "B-ISDN Protocol Reference Model and its Application", International Telecommunication Union, Telecommunication Standardization Sector, 1991.

[I.361] ITU-T Recommendation I.361: "B-ISDN ATM-Layer Specification", International Telecommunication Union, Telecommunication Standardization Sector, 1993.

[I.362] ITU-T Recommendation I.362: "B-ISDN ATM-Layer (AAL) Functional Description", International Telecommunication Union, Telecommunication Standardization Sector, 1990.

[I.363] ITU-T Recommendation I.363: "B-ISDN ATM-Layer (AAL) Specification", International Telecommunication Union, Telecommunication Standardization Sector, 1993.

Lebenslauf

von

Frank Balbach

Geburtstag, Geburtsort:	18. August 1966, in Ilshofen	
Familienstand:	verheiratet	
Schulausbildung:	1973 - 1977	Grundschule in Schwäbisch Hall
	1977 - 1979	Orientierungsstufe am Schulzentrum West in Schwäbisch Hall
	1979 - 1986	Gymnasium am Schulzentrum West in Schwäbsich Hall
Wehrdienst:	1986 - 1987	Grundwehrdienst im Stabsdienst
Hochschulausbildung:	1987 - 1993	Studium der Informatik an der Universität Erlangen-Nürnberg mit Schwerpunkt Rechnerarchitektur, Abschluß mit Diplomprüfung
Berufstätigkeit:	1993 - 1998	wissenschaftlicher Mitarbeiter am Institut für Mathematische Maschinen und Datenverarbeitung III (IMMD 3) der Universität Erlangen-Nürnberg
	seit 1998	Software-Development ACCESS bei Lucent Technologies GmbH in Nürnberg

Frank Balbach
Erlangen, den 27.01.2000

Frank Balbach

Schriftenverzeichnis über die bisherigen eigenen Veröffentlichungen

[1] Altmann, J.; Balbach, F.; Hein, A.: „An Approach for Hierarchical System Level Diagnosis of Massively Parallel Computers Combined With a Simulation-Based Method for Dependability Analysis“, In: *IEEE 1st European Dependable Computing Conference,* S. 271-285, Berlin, Germany, Oktober 1994.

[2] Altmann, J.; Balbach, F.: „Self-Checking C-net SW Control Net Diagnosis“, Universität Erlangen, Esprit 6731 - FTMPS Report 2.3.4, 1994.

[3] Vounckx, J.; Deconinck, G.; Lauwereins, R.; Viehöver, G.; Wagner, R.; Madeira, H.; Silva, J.G.; Balbach, F.; Altmann, J.; Bieker, B.; Willeke, H.: „The FTMPS-Project: Design and Implementation of Fault-Tolerance Techniques for Massively Parallel Systems“, In: *Proceedings of the HPCN Conference, Lecture Notes in Computer Science 797,* Springer Verlag, S. 401-406, München, Deutschland, 18-20th April 1994.

[4] Deconinck, G.; Vounckx, J; Cuyvers, R.; Lauwereins, R.; Bieker, B.; Willeke, H.; Maehle, E.; Hein, A.; Balbach, F.; Altmann, J.; Dal Cin, M.; Madeira, H.; Silva, J.G.; Wagner, R.; Viehöver, G.: „Fault-Tolerance in Massively Parallel Systems“, In: *Transputer Communications*, Vol 2(4), S. 241-257, Dezember 1994.

[5] Dal Cin, M.; Altmann, J.; Balbach, F.: „On the Realization of a Fault Tolerance Concept for Massively Parallel Systems“, *Interner Bericht Nr. 7/95,* Universität Erlangen, IMMD III, 1995.

[6] Deconinck, G.; Vounckx, J.; Lauwereins, R.; Altmann, J.; Balbach, F.; Dal Cin, M.; Silva, J.G.; Madeira, H.; Bieker, B.; Maehle, E.: „A Scalable Implementation of Fault Tolerance for Massively Parallel Systems“, In: *Proceedings of the 2nd Int. Conference on Massively Parallel Computing Systems (MPCS96),* Italien, 6.-9. Mai, 1996.

[7] Balbach, F.; Altmann, J.; Bieker, B.; Carreira, J.; Costa, D.; Deconinck, G.; Grigg, A.; Hein, A.; Vounckx, J.; Wenkeback, G.: „On the Realization of a Fault Tolerance Concept for Massively Prallel Systems“ In: *Proceedings of the Workshop Run by FTMPS Consortium (ESPRIT 6731) within HPCN -Europe,* 1996.

[8] Sieh, V.; Tschäche, O.; Balbach F.: „VHDL-based Fault Injection with VERIFY“ *Interner Bericht Nr. 5/96,* Universität Erlangen, IMMD III, 1996.

[9] Balbach, F.: „VHDL-basierte Fehleranalyse eines ATM-Switches“, In: *Simulationsbasierte Zuverlässigkeitsanalyse, Interner Report Nr. 6/96,* Universität Erlangen, IMMD III, 1996

[10] Sieh, V.; Tschäche, O.; Balbach, F.: „Comparing Different Fault Models Using VERIFY“, In: *Proceedings 6th Conference on Dependable Computing for Critical Applications (DCCA-6),* Grainau, Deutschland, 5.-7. März 1997, S. 59-76.

[11] Sieh, V.; Balbach, F.; Tschäche, O.: „VERIFY: Zuverlässigkeitsanalyse unter Verwendung von VHDL-Modellen mit integrierter Fehlerbeschreibung“, In: *Tagungsband 9. Workshop „Testmethoden und Zuverlässigkeit von Schaltungen und Systemen“*, Universität Bremen, Bremen, Deutschland, 9.-11. März 1997, S. 39-42.

[12] Sieh, V.; Tschäche, O.; Balbach, F.: „System Dependability Analysis using VHDL Models with Integrated Fault Descriptions“, In: *Extended Abstracts 8th European Workshop on Dependable Computing (EWDC-8),* Göteborg, Schweden, 1.-4. April 1997.

[13] Sieh, V.; Tschäche, O.; Balbach, F.: „VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions“, In: *Proceedings 27th Symposium on Fault Tolerant Computing (FTCS-27),* Seattle (WA), USA, 25.-27. Juni 1997, S. 32-36.