

# **Effiziente Erstellung und Auswertung von Rechnermodellen zur detaillierten Zuverlässigkeitsanalyse**

Der Technischen Fakultät der  
Universität Erlangen-Nürnberg

zur Erlangung des Grades

**DOKTOR-INGENIEUR**

vorgelegt von

**Volkmar Sieh**

Erlangen 1998

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

Tag der Einreichung: 04.12.97

Tag der Promotion: 12.03.98

Dekan: Prof. Dr. G. Herold

Berichterstatter: Prof. Dr. M. Dal Cin

Prof. Dr. W. H. Glauert

# Danksagung

Ich möchte mich ganz herzlich bei Prof. Dr. Mario Dal Cin für die Betreuung und Begutachtung meiner Arbeit sowie bei Prof. Dr. Wolfram Glauert für die bereitwillige Übernahme des Zweitgutachtens bedanken.

Danken möchte ich weiterhin allen Kolleginnen und Kollegen für ihre Freundschaft, Diskussionsbereitschaft, Anregungen und konstruktive Kritik.

Diese Arbeit wurde im Rahmen des Sonderforschungsbereich 182 „Multiprozessoren- und Netzwerkkonfigurationen“ von der Deutschen Forschungsgemeinschaft unterstützt.



# Kurzfassung

Werden Computer-Systeme in der Praxis eingesetzt, so muß grundsätzlich auch mit dem Versagen dieser Systeme gerechnet werden. Ihr korrektes Verhalten kann – wie bei anderen technischen Geräten auch – durch Störeinflüsse von außen oder durch Alterung beeinträchtigt sein. Werden die Systeme in kritischen Bereichen eingesetzt (z.B. zur Steuerung von Produktionsstraßen oder als ABS-Steuergerät), können ihre Ausfälle jedoch zu hohen Kosten führen oder ein großes Risiko für die Gesundheit der beteiligten Personen bedeuten.

Daher wird man versuchen, zuverlässige Systeme für derartige Einsatzgebiete zu verwenden. Schwierig ist es jedoch, die Zuverlässigkeit von Rechnern zu bestimmen. Die verwendeten Maße (z.B. die Überlebenswahrscheinlichkeit, Sicherheit, Verfügbarkeit) sind alle statistischer Natur. Die für die Statistik notwendigen großen Stichproben stehen jedoch nicht zur Verfügung. Während der Design-Phase der Systeme existiert noch keine Informationen über die Fehler der projektierten Hardware. Auch anhand eines Prototyps sind statistisch aussagefähige Daten über die Fehler eines Systems und ihre Auswirkungen nicht schnell genug zu ermitteln, da die Fehler in diesem Fall zu selten auftreten.

Es ist daher notwendig, die Zuverlässigkeit durch die Analyse von Modellen dieser Systeme abzuschätzen. Dazu müssen zunächst alle die Zuverlässigkeit betreffenden Daten in einer Beschreibung zusammengefaßt werden, die nachfolgend ausgewertet wird. Wie in der Literatur beschriebene und eigene Experimente gezeigt haben, können kleine Unterschiede in den Modellen jedoch schon zu großen Änderungen in der berechneten Zuverlässigkeit führen. Daher ist es unbedingt erforderlich, detailreiche Modelle einfach – und damit fehlerfrei – erstellen zu können.

In der vorliegenden Arbeit wird eine Erweiterung der Modellierungssprache VHDL vorgestellt, die es auf einfache Weise erlaubt, sowohl die deterministischen Eigenschaften der Hardware als auch die stochastischen Attribute der Fehler (wie z.B. Fehlerrate, mittlere Fehlerdauer) zu beschreiben. Im Gegensatz zu aus der Literatur bekannten Verfahren besitzen diese Modelle den Vorteil, daß sie in sich abgeschlossen sind. Es entfallen damit alle Probleme, die auftreten, wenn die Spezifikationen der einzelnen, sich gegenseitig beeinflussenden Eigenschaften des Systems auf verschiedene Beschreibungsdateien verteilt sind. Wie Experimente gezeigt haben, können mit dem vorgestellten Ansatz deutlich größere und detailliertere Modelle als bisher möglich fehlerfrei erstellt werden, wodurch eine genauere Abschätzung der Zuverlässigkeit eines Systems erreichbar ist.

Sehr detaillierte Modelle können aufgrund ihrer Komplexität nur noch in Ausnahmefällen mit analytischen Methoden ausgewertet werden. Daher werden die modellierten Systeme simuliert und verschiedene, zufällig ausgewählte Fehlerszenarien durchgespielt. Anhand der ermittelten Resultate und der im Modell beschriebenen Eigenschaften des Systems werden dann statistische Aussagen über das System getroffen. Leider ist für eine aussagekräftige Statistik eine große Stichprobe notwendig. Das heißt, daß eine Vielzahl verschiedener Fehlerszenarien durchgetestet werden muß. Jedes einzelne Fehlersimulationsexperiment ist jedoch sehr zeitaufwendig. Daher werden eine Reihe von Möglichkeiten entwickelt und bewertet, Fehlersimulationsexperimente zu beschleunigen. Mit diesen Ansätzen ist es möglich, die Simulationsdauer um mehrere Größenordnungen zu verringern und damit die Auswertung von detaillierten Modellen in der Praxis überhaupt durchzuführen.

Die entwickelte Modellierungssprache sowie alle beschriebenen Verfahren zur Modellauswertung sind im Modellierungswerkzeug „VERIFY“ (VHDL-based Evaluation of Reliability by Injecting Faults efficiently) verwirklicht und getestet.



# Inhaltsverzeichnis

<b>1. Einleitung</b> . . . . .	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Stand der Technik. . . . .	3
1.3 Ziel der Arbeit . . . . .	3
1.4 Übersicht . . . . .	4
<b>2. Modellaufbau</b> . . . . .	<b>5</b>
2.1 Bestehende Modellierungsverfahren . . . . .	6
2.1.1 Virtuelle Systeme . . . . .	6
2.1.2 Instrumentierte Systeme . . . . .	7
2.2 Fehlermodell mit Zeitparametern. . . . .	9
2.3 Trennung von Modell und Modellierungswerkzeug . . . . .	10
2.4 Verschmelzung von System- und Fehlermodell . . . . .	11
2.5 Fehlermodelle auf niedriger Abstraktionsebene . . . . .	13
2.6 VHDL-Erweiterung. . . . .	17
2.6.1 Syntaktische und semantische Ergänzungen . . . . .	17
2.6.2 Beispiel . . . . .	19
2.6.3 Bewertung. . . . .	20
<b>3. Effiziente Experimentdurchführung</b> . . . . .	<b>23</b>
3.1 Bestehende Simulationsverfahren . . . . .	23
3.1.1 Simulation virtueller Systeme . . . . .	24
3.1.2 Simulation instrumentierter Systeme . . . . .	26
3.2 Einzelfehlerannahme . . . . .	27
3.2.1 Fehlerinjektion gemäß der relativen Häufigkeit der Fehler. . . . .	28
3.2.2 Fehlerinjektion mit hoher Konfidenz der Ergebnisse. . . . .	29
3.2.3 Praktische Durchführung der Experimente . . . . .	32
3.3 Simulation mit Hilfe des Zielsystems. . . . .	33
3.3.1 Prinzipielles Verfahren . . . . .	33
3.3.2 Bewertung des Verfahrens . . . . .	35
3.4 Multi-Threaded Fault-Injection. . . . .	36

3.4.1	Prinzipielles Verfahren . . . . .	36
3.4.2	Bewertung des Verfahrens . . . . .	37
3.5	Parallele Simulation . . . . .	37
3.5.1	Prinzipielles Verfahren . . . . .	37
3.5.2	Bewertung des Verfahrens . . . . .	38
3.6	Vergleich mit Golden-Run . . . . .	39
3.6.1	Prinzipielles Verfahren . . . . .	39
3.6.2	Zustandsvergleich durch Signaturen . . . . .	39
3.6.3	Inkrementeller Zustandsvergleich . . . . .	40
3.6.4	Bewertung der Verfahren. . . . .	41
3.7	Dynamisches Wechseln von Modellen. . . . .	42
3.7.1	Probleme beim dynamischen Wechseln von Modellen . . . . .	42
3.7.2	Automatische Generierung schneller simulierbarer Modelle . . . . .	44
3.7.3	Bewertung . . . . .	48
3.8	Irrelevante Fehler in Registern . . . . .	49
3.8.1	Beschreibung des Verfahrens . . . . .	49
3.8.2	Bestimmung der Schreib- und Lesezugriffe . . . . .	51
3.8.3	Beispiel . . . . .	52
3.8.4	Bewertung des Verfahrens . . . . .	58
3.9	Irrelevante Fehler in kombinatorischen Schaltungen . . . . .	59
3.9.1	Prinzipielles Verfahren . . . . .	59
3.9.2	Bewertung des Verfahrens . . . . .	62
<b>4.</b>	<b><i>Detaillierte Experimentauswertung</i></b> . . . . .	<b>65</b>
4.1	Bestehende Auswerteverfahren . . . . .	67
4.1.1	Simulation virtueller Systeme . . . . .	67
4.1.2	Simulation instrumentierter Systeme. . . . .	68
4.2	Zeitverschiebung durch Fehler . . . . .	69
4.2.1	Darstellung des Problems . . . . .	69
4.2.2	Algorithmus zur Bestimmung der Zeitverschiebung. . . . .	70
4.2.3	Bewertung . . . . .	72
4.3	Zustandsänderung durch Fehler . . . . .	73
4.4	Ausgabeänderung durch Fehler . . . . .	73

4.5	Dynamische Anpassung der Zeitauflösung . . . . .	75
4.5.1	Prinzipielles Verfahren . . . . .	75
4.5.2	Bewertung des Verfahrens . . . . .	77
<b>5.</b>	<b>Modellierungsumgebung VERIFY . . . . .</b>	<b>79</b>
5.1	Modellerstellung . . . . .	79
5.2	Compilierung . . . . .	80
5.3	Simulation. . . . .	80
5.4	Auswertung . . . . .	81
<b>6.</b>	<b>Fallbeispiel DP32 . . . . .</b>	<b>83</b>
6.1	Modellerstellung . . . . .	83
6.2	Compilierung, Simulation und Auswertung . . . . .	87
6.3	Ergebnisse. . . . .	88
6.3.1	Beschreibung der Diagrammdarstellung . . . . .	89
6.3.2	Ergebnisse. . . . .	91
<b>7.</b>	<b>Zusammenfassung und Ausblick . . . . .</b>	<b>101</b>
7.1	Zusammenfassung . . . . .	101
7.2	Ausblick. . . . .	102
	<b>Literatur . . . . .</b>	<b>105</b>
	<b>Indexregister . . . . .</b>	<b>113</b>
	<b>Schriftenverzeichnis . . . . .</b>	<b>117</b>
	<b>Lebenslauf . . . . .</b>	<b>119</b>



# Abbildungsverzeichnis

Abb. 1: Generelles Korrespondenzproblem von System- und Fehlermodell . . . . .	12
Abb. 2: Verschmolzenes System- und Fehlermodell . . . . .	13
Abb. 3: Korrespondenzproblem von System- und Fehlermodell auf Verhaltensebene. .	14
Abb. 4: Modellerstellung ohne durch eine Synthese eingeführte Korrespondenzprobleme	15
Abb. 5: Fehlersimulation auf Layout-Ebene . . . . .	16
Abb. 6: Regel für Signal-Deklaration in VHDL-93. . . . .	18
Abb. 7: Regel für Signal-Deklaration im erweiterten VHDL. . . . .	19
Abb. 8: NOT-Gatter: Entity-Beschreibung . . . . .	19
Abb. 9: NOT-Gatter: Verhaltensmodell ohne Fehlermodell . . . . .	19
Abb. 10: NOT-Gatter: Verhaltensmodell mit integriertem Fehlermodell . . . . .	20
Abb. 11: Signalisieren von Ereignissen in VHDL . . . . .	24
Abb. 12: Warten auf Ereignisse in VHDL . . . . .	24
Abb. 13: Abfragen von Ereignissen in VHDL . . . . .	25
Abb. 14: Algorithmus zur zufälligen Auswahl eines Fehlers . . . . .	29
Abb. 15: Simulation mit Hilfe des Zielsystems . . . . .	34
Abb. 16: Multi-Threaded Fehlerinjektion . . . . .	36
Abb. 17: Master-Knoten berechnet Golden-Run. . . . .	38
Abb. 18: Jeder Knoten berechnet Golden-Run für sich . . . . .	38
Abb. 19: Automatische Generierung einfacher und detailreicher Modelle . . . . .	43
Abb. 20: XOR-Gatter aus einzelnen Invertern und NAND-Gattern . . . . .	44
Abb. 21: VHDL-Entity-Beschreibung eines XOR-Gatters . . . . .	45
Abb. 22: VHDL-Code für XOR-Gatter (nicht optimiert) . . . . .	45
Abb. 23: VHDL-Code für XOR-Gatter (Prozeß-optimiert) . . . . .	46
Abb. 24: Struktogramm: Ermittlung der Berechnungsreihenfolge. . . . .	47
Abb. 25: VHDL-Entity-Beschreibung eines 32-Bit-Registers . . . . .	47
Abb. 26: VHDL-Beschreibung eines 32-Bit-Registers (Gatterebene, nicht optimiert) . .	48
Abb. 27: VHDL-Beschreibung eines 32-Bit-Registers (Gatterebene, Prozeß-optimiert) .	48
Abb. 28: Best-Case bei der Verschmelzung von Prozessen . . . . .	48

Abb. 29: Worst-Case bei der Verschmelzung von Prozessen . . . . .	49
Abb. 30: Lebenszyklen von Zuständen . . . . .	50
Abb. 31: Intervalle, in denen ein speicherndes Element keine relevanten Daten enthält . . .	50
Abb. 32: Erforderliche Mikroinstruktionen zur Ausführung einer Addition . . . . .	51
Abb. 33: Register oder Latch mit nachfolgenden Gattern . . . . .	52
Abb. 34: Compiler-Abhängigkeit der Länge der Zugriffsintervalle (Register-File). . . . .	54
Abb. 35: Compiler-Abhängigkeit der Länge der Zugriffsintervalle (Stack-Segment) . . . .	55
Abb. 36: Hauptschleife des busy-Programms (C-Code und M88100-Assembler) . . . . .	55
Abb. 37: Benchmark-Abhängigkeit der Länge der Zugriffsintervalle (Register-File) . . . .	57
Abb. 38: Benchmark-Abhängigkeit der Länge der Zugriffsintervalle (Stack-Segment) . . .	58
Abb. 39: Phasen, in denen Fehler Auswirkungen haben. . . . .	60
Abb. 40: Berechnung der kritischen Fehler . . . . .	61
Abb. 41: Normale und fehlerhafte Zustandsübergänge in zyklisch arbeitenden Systemen .	65
Abb. 42: Normale und fehlerhafte Zustandsübergänge in einmalig arbeitenden Systemen .	66
Abb. 43: Durch Fehler hervorgerufene Zeitverschiebung . . . . .	70
Abb. 44: Algorithmus A zur Berechnung der durch Fehler provozierten Zeitverschiebung	71
Abb. 45: Algorithmus B zur Berechnung der durch Fehler provozierten Zeitverschiebung	72
Abb. 46: Ausgabenprotokollierung mit Hilfe der MMU . . . . .	75
Abb. 47: Zeitpunkt eines Vergleiches in Abhängigkeit von seiner laufenden Nummer. . . .	76
Abb. 48: Zuordnung der Meßzeitpunkte zu den Ereigniszeitpunkten . . . . .	77
Abb. 49: Modellerstellung, Simulation und Auswertung mit VERIFY . . . . .	79
Abb. 50: DP32 RISC-Prozessor . . . . .	84
Abb. 51: Testprogramm für DP32 . . . . .	86
Abb. 52: Beispieldiagramm . . . . .	89
Abb. 53: Unterteilung der Recovery-Zeiten nach Fehlerorten (bereits behobene Fehler) . . . . .	90
Abb. 54: Unterteilung der Recovery-Zeiten nach Fehlerorten (noche zu korrigierende Fehler). . . . .	91
Abb. 55: Verteilung der Recovery-Zeiten des „Simple“-Modells	

bei Stuck-At-Fehlern mit Unterteilung der bereits korrigierten Fehler . . . . .	92
Abb. 56: Verteilung der Recovery-Zeiten des „Simple“-Modells	
bei Stuck-At-Fehlern mit Unterteilung der noch zu korrigierenden Fehler . . . . .	92
Abb. 57: Verteilung der Recovery-Zeiten des „Advanced“-Modells	
bei Stuck-At-Fehlern mit Unterteilung der bereits korrigierten Fehler . . . . .	93
Abb. 58: Verteilung der Recovery-Zeiten des „Advanced“-Modells	
bei Stuck-At-Fehlern mit Unterteilung der noch zu korrigierenden Fehler . . . . .	94
Abb. 59: Verteilung der Recovery-Zeiten des Modells	
bei Bit-Flip-Fehlern mit Unterteilung der bereits korrigierten Fehler . . . . .	95
Abb. 60: Verteilung der Recovery-Zeiten des Modells	
bei Bit-Flip-Fehlern mit Unterteilung der noch zu korrigierenden Fehler . . . . .	95
Abb. 61: Verteilung der Recovery-Zeiten bei Pin-Level-Fehlern . . . . .	96
Abb. 62: Recovery-Zeit der verschiedenen Hardware- und Fehlermodelle (0-1ns) . . . . .	97
Abb. 63: Recovery-Zeit der verschiedenen Hardware- und Fehlermodelle (0-2000ns) . . . . .	98



# Verzeichnis der Tabellen

Tabelle 1: Verschiedene Systemmodellierungsebenen und ihre Fehlermodelle . . . . .	11
Tabelle 2: Beschleunigungsfaktoren durch Vergleich mit Golden-Run. . . . .	41
Tabelle 3: Mittlerer Nutzungsgrad von Speicherplätzen (Compiler-Abhängigkeit) . . .	53
Tabelle 4: Mittlerer Nutzungsgrad von Speicherplätzen (Benchmark-Abhängigkeit) . .	56
Tabelle 5: Vergleich von Nutzungsgrad und Fehlerauswirkungswahrscheinlichkeit . . .	57
Tabelle 6: Verbesserung der Genauigkeit durch Berechnung der Zeitverschiebung . . .	73
Tabelle 7: Hardware- und Fehlermodell-Kombinationen . . . . .	89
Tabelle 8: Anzahl der verschiedenen Fehlermöglichkeiten . . . . .	97



# 1. Einleitung

## 1.1 Motivation

Im gegenwärtigen Informationszeitalter werden immer mehr Prozesse durch Computer gesteuert. Von kleinen, elektronischen Spielen bis hin zu riesigen Anlagen in den Fabriken, der Energieerzeugung und im Verkehr läßt sich der Computer aus der heutigen Welt nicht mehr wegdenken. Er hat seinen Einsatz in nahezu allen Bereichen des modernen Lebens in den Industriestaaten gefunden. Da die Kosten der elektronischen Bauteile wie zum Beispiel Mikrocontroller, Speicher-Chips usw. vermutlich auch in den nächsten Jahren weiter fallen und die Leistungsfähigkeit dieser Bauteile weiter steigen werden, kann man auch für die Zukunft mit einem wachsenden Einsatzgebiet dieser „intelligenten“ Steuerungen rechnen.

Durch den Einsatz von Rechnern in immer mehr, immer komplexeren und wichtigeren Systemen entsteht auch eine zunehmende Abhängigkeit von deren einwandfreier Funktion. Je umfangreicher die Steuerungen jedoch werden, desto größer wird auch die Gefahr, daß einzelne Komponenten ausfallen und das Gerät seine Funktion nicht mehr erfüllen kann. Dies kann je nach Einsatzgebiet des Gerätes unter Umständen katastrophale Folgen haben. Man denke zum Beispiel an Ausfälle von Steuerungen in Fahrstühlen, Kraftfahrzeugen, Flugzeugen oder Kernkraftwerken. Hier wird man immer ein „fail-safe“-Verhalten der Systeme fordern, d.h. daß sie im Falle interner Fehler automatisch in einen sicheren Zustand übergehen und dort verbleiben, bis die Fehler behoben sind.

In anderen Bereichen kann zwar ein kurzzeitiges Fehlverhalten von Anlagen toleriert werden, deren langfristiger Stillstand würde jedoch zu großen Kosten führen. Fällt beispielsweise eine Montagestraße einer Fabrik für mehrere Tage aus, kann dies im Extremfall zum Konkurs der beteiligten Firmen führen. Daher fordern Käufer derartiger Systeme immer häufiger Garantien für den ununterbrochenen bzw. nur kurz unterbrochenen, fehlerfreien Betrieb der Systeme.

Auch die Zuverlässigkeit von reinen Computer-Systemen für Organisations-, Konstruktions- und Simulationsaufgaben gewinnt heutzutage immer mehr an Bedeutung. Nahezu kein Produkt wird ohne Computereinsatz konstruiert, kein Betrieb mehr ohne Computer geführt, keine Zeitung ohne Layout-Programm gesetzt, keine Banküberweisung mehr per Hand ausgeführt. Alle Benutzer gehen jedoch praktisch jederzeit von einer korrekten Funktion dieser Systeme aus, obwohl bekannt ist, daß elektronische Bauteile – wie mechanische auch – mit der Zeit altern und ausfallen können. Aufgrund der großen Anzahl von Transistoren, Leitungen usw. eines Computer-Systems ist der Ausfall einer Komponente normalerweise nicht sofort bemerkbar. Das System wird sich auf den ersten Blick weiterhin richtig verhalten und nur aufgrund spezieller Tests als fehlerhaft erkannt werden können. Es besteht daher immer die Gefahr, daß unvorsichtige Benutzer dieser Systeme durch deren Fehlfunktionen Daten verlieren oder daß sie aufgrund von Fehlinformationen des Computers zu falschen Schlüssen verleitet werden.

Da sich Ausfälle von einzelnen Komponenten jedoch nie ausschließen lassen, kommt den Fehlertoleranzeigenschaften derartiger Systeme eine wachsende Bedeutung zu. Die Rechner müssen so aufgebaut sein, daß sie auch nach dem Ausfall von Komponenten nach außen hin fehlerfrei weiterarbeiten – zumindest solange, bis ein Service-Techniker die Möglichkeit hatte, die defekten Komponenten auszutauschen. Dies wird man zwar auch in Zukunft niemals mit Sicherheit garantieren können, jedoch ist es durch den Einsatz von Zeit-, Hardware- und Software-Redundanz möglich, die Wahrscheinlichkeit für ein solches Verhalten zu verbessern.

## 1. Einleitung

---

Leider sind jegliche Fehlertoleranzmaßnahmen mit Kosten verbunden. Es entstehen Kosten beim Design, da die Systeme durch den Einsatz von Redundanz komplexer werden. Neben den Entwurfskosten steigen außerdem die Herstellungskosten der Systeme, da diese zusätzliche Ersatzkomponenten, überprüfende Einheiten usw. enthalten, die der Fehlertoleranz dienen. Auch die Zeitredundanz kostet etwas: Leistung. Gegenwärtig sind fehlertolerante Systeme häufig ein Vielfaches teurer als normale, nicht fehlertolerante Computer mit ansonsten gleicher Rechenleistung. Daher muß immer eine Kosten-/Nutzenanalyse gezielt zwischen den möglichen Alternativen abwägen.

Schwierig ist es zur Zeit aber noch, den Nutzen der Fehlertoleranzmaßnahmen zu bestimmen. Zwar ist in der Literatur inzwischen eine Vielzahl von verschiedenen Möglichkeiten bekannt, Fehlern mit ausgeklügeltem Hardware- und Software-Einsatz zu begegnen (Selbsttests wie z.B. Watchdog-Prozessoren ([Mahmood88], [Michel92], [Ohlsson92], [Hönig94]), Master-Checker-Konfigurationen [Motorola90a], [Motorola90b], [Gaisler97], Check-Summen (CRC, Parity), algorithmenbasierte Checks (einen Überblick gibt z.B. [Böhm94]), es ist jedoch meistens nicht bekannt, welche Fehler damit genau toleriert werden können und wie häufig derartige Fehler überhaupt auftreten. Daß jede Hardware-Redundanz, die aus Fehlertoleranzgründen eingeführt wird, selbst wieder fehlerhaft sein kann, erschwert die Analyse der Fehlertoleranzeigenschaften zusätzlich.

Im Falle der sogenannten „fail-safe“-Systeme stellt sich zwar nicht die Frage, ob die Steuerungen einen bestimmten Fehler tolerieren – wenn sie wirklich „fail-safe“ sind, geraten sie niemals in einen kritischen Zustand –, dafür ist aber eine Abschätzung äußerst wichtig, wie häufig diese Systeme aufgrund interner Fehler in den sicheren, aber nicht produktiven Zustand übergehen werden, da dann der durch das System angestrebte Nutzen entfällt. Die Aussage „das System wird nie in einen kritischen Zustand übergehen“ allein reicht nicht aus. Eine Abschätzung, wie häufig es in den nicht produktiven Sicherheitszustand übergeht, ist ebenso wichtig.

Es existiert eine Vielzahl von Faktoren, die die Verlässlichkeit eines Systems beeinflussen. Dazu gehört unter anderem auch die äußere Umgebung. So können die Temperatur, die Temperaturschwankungen denen ein System ausgesetzt ist, elektromagnetische Wellen, Schwerionen, Alpha-Teilchen oder Schwankungen der Speisespannung von außen auf den Zustand des Systems einwirken. Wie stark sind diese Einflüsse? Zu den Faktoren zählen weiterhin die Qualität der inneren Komponenten des Systems. Wie leicht lassen sie sich durch äußere Einflüsse kurzzeitig in ihrem Verhalten ändern oder gar zerstören? Eine weitere Frage ist, ob sich von außen injizierte Fehler zu einem vom Originalzustand abweichenden Zustand manifestieren und ob sich dies in einem Fehlverhalten oder sogar Ausfall des Systems auswirkt. Dies ist sehr stark von der Struktur und der Rechenlast des Systems abhängig. Relativ kleine Änderungen in der Hard- und Software können zu großen Änderungen bei den Fehlertoleranzeigenschaften eines Systems führen.

Die Bewertung von Fehlertoleranzmaßnahmen ist daher eine Aufgabe, die sehr sorgfältig durchgeführt werden muß, soll ein Analyseergebnis gewonnen werden, das die Realität korrekt widerspiegelt. Der Aufwand ist jedoch unbedingt notwendig, wenn eine sinnvolle Kosten-/Nutzenanalyse von Fehlertoleranzmaßnahmen gefordert ist. Je größer die Anforderungen der Anwendung an die Zuverlässigkeit des Systems werden, desto wichtiger wird auch diese Analyse. Ansonsten enthält ein System zuviel Redundanz und die Kosten steigen unnötig an, oder es mangelt an Fehlertoleranzmaßnahmen mit unter Umständen katastrophalen Auswirkungen für die Benutzer.

## 1.2 Stand der Technik

Eine Bewertung der Fehlertoleranzeigenschaften ist – wie im vorigen Absatz beschrieben – ein wichtiger Teil des Entwurfsprozesses neuer Hardware-Komponenten für digitale Computer-Systeme. Wenn möglich, sollte ein Designer bei dieser Aufgabe durch gesicherte Methoden und geeignete Werkzeuge in seiner Arbeit unterstützt werden. Wie sich bei Gesprächen mit Vertretern von Entwicklungsabteilungen der Industrie immer wieder herausgestellt hat, existieren zur Zeit keine allgemein akzeptierten Methoden zur Bewertung von Fehlertoleranzeigenschaften bezüglich spontan auftretender Hardware-Fehler. Zwar werden z.B. Fehlerbäume, Petri-Netze, Markov-Ketten und Fehlerinjektionsverfahren eingesetzt, sie sind jedoch nicht in den eigentlichen Design-Prozeß eingebettet. Diese Methoden erfordern daher immer zusätzliche Entwicklungskosten und sind sehr anfällig für Handhabungsfehler.

Sie bieten zudem nur die Möglichkeit zu überprüfen, ob bestimmte Systeme „fail-safe“ sind, sowie Möglichkeiten, die Ausfallrate des Gesamtsystems grob abzuschätzen. Dies gilt jedoch nur bezüglich permanenter, d.h. nach ihrem erstmaligen Auftritt dauerhafter Fehler. Reaktionen auf temporäre Fehler, die nur kurz aufgrund von äußeren Störungen auftreten, können von bestehenden Werkzeugen nur exemplarisch überprüft werden. Interessante Größen wie z.B. die Rate, mit der ein System zwar temporär durch Fehler in seinem Verhalten gestört, aber nicht zum vollständigen Ausfall gebracht wird, die mittlere Ausfalldauer und die Rate, mit der aufgetretene Fehler maskiert, d.h. nicht nach außen sichtbar werden, können nicht abgeschätzt werden. Man vermutet jedoch, daß temporäre Fehler um Größenordnungen häufiger auftreten als permanente [Iyer86][Siewiorek82].

Da die analytischen Verfahren (Fehlerbäume [Lee85], Petri-Netze [Marsan91], [Marsan95], Markov-Modelle [Trivedi82], [Buchholz94], [Stewart94]) aufgrund des ansonsten zu großen Rechenaufwandes nur für einfache Modelle geeignet sind, wird im folgenden nur auf simulative Ansätze eingegangen. Ihre Eigenschaften sowie ihre Vor- und Nachteile werden in den Abschnitten 2.1 („Bestehende Modellierungsverfahren“), 3.1 („Bestehende Simulationsverfahren“) und 4.1 („Bestehende Auswerteverfahren“) näher beschrieben.

## 1.3 Ziel der Arbeit

Ziel dieser Arbeit ist es daher, die Bewertung von Fehlertoleranzmaßnahmen – sowohl bezüglich permanenter als auch bezüglich temporärer Fehler – in den Design-Prozeß der Systeme mit einzubeziehen. Wie z.B. Leistungsanalysen oder Funktionstests sollen auch Fehlertoleranzanalysen während der Design-Phase ohne großen Zusatzaufwand möglich sein. Ist eine Beschreibung eines neuen Systems erstellt, soll eine Bewertung der Eigenschaften gegenüber Hardware-Fehlern soweit wie möglich automatisch generiert werden können.

Um jedoch Fehlertoleranzaspekte in den Entwurfsprozeß digitaler Systeme zu integrieren, müssen eine Reihe von Änderungen in diesem Prozeß vorgenommen werden. So hat eine solche Integration schließlich Auswirkungen auf den Modellierungsvorgang und die Art der Simulation bzw. Analyse dieser Modelle.

Ein weiterer, wichtiger Punkt dieser Arbeit ist die effiziente Auswertung der Rechnermodelle. Viele bekannte Verfahren erfordern soviel Rechenzeit, daß sie allein aus diesem Grund keine generelle Akzeptanz finden. Nur wenn Modelle innerhalb relativ kurzer Zeit mit genügender Genauigkeit ausgewertet werden können, kann mit einem breiten Einsatz in der Praxis gerechnet werden. Schließlich ist – gerade in der Computer-Industrie – Zeit gleich Geld.

### 1.4 Übersicht

Das nächste Kapitel („Modellaufbau“) enthält Überlegungen, wie Modelle aussehen müssen, die zum einen als Spezifikation für die Herstellung von digitalen Schaltungen dienen und zum anderen auch für die Zuverlässigkeitsanalyse verwendet werden können. Es werden die aktuell verwendeten Modellierungsverfahren und deren Vor- und Nachteile beschrieben. Im Anschluß daran wird ein Ansatz vorgestellt, der die aufgezeigten Probleme vermeidet und der deutlich genauere Ergebnisse neben einer besseren Handhabbarkeit der Modellierungssprache verspricht.

Gegenstand der Untersuchungen im 3. Kapitel („Effiziente Experimentdurchführung“) ist, wie bestehende Modelle möglichst effizient simuliert werden können. Zunächst werden die bisher verwendeten Verfahren mit ihren Vor- und Nachteilen vorgestellt. Darauf folgen eine Reihe von Vorschlägen, wie Fehlersimulationen beschleunigt werden können. Neben der Vorstellung der einzelnen Verfahren finden sich auch Beispiele mit Meßwerten sowie theoretische Effizienzanalysen.

Die von den Fehlersimulationen generierten Spuren müssen nach Zuverlässigkeitskriterien ausgewertet werden. So sollen beispielsweise die Ausfallraten, die Verteilung der Ausfalldauern usw. ohne Hilfe des Benutzers berechnet werden. Der erste Teil des Kapitels 4 („Detaillierte Experimentauswertung“) stellt die aus der Literatur bekannten Verfahren zur vollautomatischen Auswertung von Simulationsexperimenten vor. Am Schluß des Kapitels finden sich Lösungen zu einigen der bisher ungelösten Probleme dieser Verfahren.

Die in den Kapiteln 2 bis 4 vorgestellten, neuen Methoden zur Bewertung von Fehlertoleranzeigenschaften wurden in ein neues Modellierungs- und Auswertewerkzeug integriert und getestet. Kapitel 5 („Modellierungsumgebung VERIFY“) gibt einen Überblick über den Aufbau dieses neuen Werkzeugs.

Kapitel 6 verwendet das im vorangehenden Kapitel beschriebene Werkzeug, um einen bekannten Prozessor bezüglich seiner Zuverlässigkeit zu bewerten. Die präsentierten Ergebnisse belegen gleichzeitig die in den Kapiteln 2 („Modellaufbau“) und 3 („Effiziente Experimentdurchführung“) angesprochenen Probleme der derzeit verwendeten Modellierungs- und Auswertewerkzeuge.

---

## 2. Modellaufbau

Um ein System bezüglich einer bestimmten Größe (z.B. Leistung, Zuverlässigkeit, Fehlerüberdeckung, o.ä.) bewerten zu können, muß es mit allen zur Auswertung notwendigen Daten eindeutig beschrieben sein. Diese Beschreibung wird als Modell bezeichnet. Sämtliche Aussagen, die eine Bewertung aufgrund dieser Beschreibung liefert, sind im allgemeinen nur für dieses ganz spezielle Modell gültig. Weicht die Beschreibung vom realen System ab, werden auch die aus dem Modell gewonnenen Daten von den eigentlichen Systemwerten abweichen. Zur Bewertung von Systemen ist es daher nötig, ein möglichst genaues Modell des Systems aufzustellen (Hinweise dazu geben z.B. [Jacomet91] oder [Khare95]). Aufgrund der großen Komplexität der in der Praxis vorkommenden Modelle ist eine maschinelle Auswertung dieser Beschreibungen erforderlich. Die Beschreibungen müssen daher eine automatische Analyse erlauben. Daraus ergeben sich zwei wesentliche Anforderungen an eine Modellierungssprache:

Die Modelle müssen

- vom Benutzer möglichst einfach erstellbar und
- von einem Analysewerkzeug möglichst schnell und möglichst genau auswertbar sein

Jedes Modell, das zur Analyse von Fehlertoleranzeigenschaften verwendet wird, kann in zwei Teile unterteilt werden: das System- und das Fehlermodell:

*Definition „Systemmodell“:*

*Das „Systemmodell“ umfaßt die Beschreibung der Hardware, der Software und der Umgebung des eigentlichen Systems. Das Systemmodell beschreibt damit das gesamte Verhalten des modellierten Systems ohne den Einfluß von Fehlern.*

*Definition „Fehlermodell“:*

*Ein „Fehlermodell“ ist eine Beschreibung, welche Fehler im System auftreten können und welche Auswirkungen sie haben. Für alle Fehler sind Wahrscheinlichkeitsverteilungen angegeben, die ihren Auftrittszeitpunkt und ihre Auftrittsdauer beschreiben. Fehlt im Fehlermodell die Angabe der Wahrscheinlichkeitsverteilungen, wird im folgenden von einem „einfachen Fehlermodell“ gesprochen.*

Weiterhin werden im folgenden die Begriffe „Verhaltensmodell“, „Strukturelles Modell“, Modell auf „Layout-Ebene“, „Schaltungsebene“, „Gatterebene“, „Register-Transfer-Ebene“ bzw. „Systemebene“ benutzt. Ihre Verwendung ist konform zu [Ashenden90] und [Bleck96].

Nur wenn das Modell vollständig beschrieben worden ist, kann es eindeutig und vollautomatisch ausgewertet werden. Fehlen notwendige Angaben, sind verschiedene Interpretationen des Modells und damit unterschiedliche Analyseergebnisse möglich. Für jedes System gilt, daß sein Verhalten von seinem internen Zustand, seinen Fehlern und seinen Eingabedaten abhängt. Soll das Verhalten eines Systems simuliert oder analysiert werden, muß daher bekannt sein, wie sich die Daten an seinen Eingabeparametern mit der Zeit ändern und welche Fehler zu welchen Zeiten aktiv sind.

*Zur Zuverlässigkeitsanalyse müssen deshalb neben der Spezifikation der Hardware eindeutige und vollständige Beschreibungen der Software, der Umgebung und der möglichen Fehler Bestandteile des auszuwertenden Modells sein.*

Der Einfluß der Software-Last auf die Simulationsergebnisse wird z.B. in [Iyer86], [Güthoff95] und auch im Abschnitt 3.8 untersucht. Aber auch die Umgebung eines System kann – z.B. durch generierte Interrupts – als Last auf die Hardware einwirken. Kapitel 6 zeigt, wie unterschiedlich

## 2. Modellaufbau

---

die Ergebnisse einer Zuverlässigkeitsuntersuchung sein können, wenn verschiedene Fehlermodelle verwendet werden. Im folgenden wird daher immer von vollständigen und eindeutigen Modellen ausgegangen.

Im Abschnitt 2.1 werden die bisher existierenden Modellierungsverfahren vorgestellt. Die gravierendsten Probleme werden in den Abschnitten 2.2 bis 2.5 näher untersucht. In 2.6 wird ein neuer Ansatz für eine Modellierungssprache vorgestellt, der sich für Bewertungen bezüglich Fehlertoleranzeigenschaften als weitaus günstiger herausgestellt hat als die bisher existierenden Ansätze.

### 2.1 Bestehende Modellierungsverfahren

Eine gute Übersicht über die zur Zeit existierenden Modellierungsverfahren für die Bewertung von Systemen hinsichtlich ihrer Fehlertoleranz bieten [Clark95] und [Iyer94]. Die verschiedenen Werkzeuge lassen sich grob in zwei Kategorien unterteilen. Die erste Gruppe bilden die Verfahren, welche die zu bewertenden Systeme nur durch Modellbeschreibungen darstellen („virtuelle Systeme“) und diese Beschreibungen dann analytisch oder simulativ auswerten. Existierende Systeme werden von der zweiten Gruppe von Werkzeugen mit Hardware- und/oder Software erweitert („instrumentierte Systeme“). Mit Hilfe der Instrumentierung lassen sich Fehler injizieren und deren Auswirkungen beobachten.

#### 2.1.1 Virtuelle Systeme

##### Modelle mit integriertem Fehlermodell

Modellierungswerkzeuge wie z.B. **REACT** [Clark93], **ADEPT** [Kumar95], **CSIM** [Saleh87], **DEPEND** [Goswami90] und **SimPar** [Hein95] bieten die Möglichkeit, aus vorgefertigten, „atomaren“ Komponenten (z.B. CPU's, Speichern, Bussen, usw.) größere Systeme „zusammenzustecken“. In den Einzelkomponenten ist jeweils das fehlerfreie und fehlerhafte Verhalten beschrieben. Die Werkzeuge verwenden für die Fehlerraten und Reparaturzeiten bestimmte Verteilungsfunktionen, die vom Benutzer parametrierbar sind.

Bedingt durch den großen Overhead der verwendeten Compiler und Simulatoren können nur Systeme aus relativ wenigen Teilkomponenten aufgebaut werden. Sie müssen daher relativ abstrakt sein (Systemebene). Detailreiche Modelle (z.B. Systeme auf Gatterebene) sind mit diesen Systemen nicht denkbar (siehe dazu auch Abschnitt 2.5).

Die Kommunikation zwischen den Einzelkomponenten (Objekte) erfolgt über Methodenaufrufe (REACT, DEPEND und SimPar). Für Hardware-Modellierung wäre jedoch eine Kommunikation über Signale (wie z.B. in VHDL<sup>1</sup> oder VERILOG) sehr viel besser geeignet (ADEPT).

##### Nachträglich um Fehlermodelle erweiterte Modelle

Existierende Systemmodelle mit Fehlerbeschreibungen zu versehen, ist im allgemeinen sehr mühsam. Daher wird in vielen Projekten versucht, Werkzeuge zu entwickeln, die bestehenden Modelle automatisch um Fehlermodelle erweitern. Die Fehlermodelle orientieren sich an den Möglichkeiten, die die verwendeten Modellierungssprachen bieten.

---

1. VHDL: Very High Speed Integrated Circuit Hardware Description Language

Als Beispiele seien hier [Armstrong92], **EMAX** [Kanawati94] und das **MEFISTO**-Werkzeug [Jenn94] erwähnt. Diese enthalten einfache, allgemeine Regeln, die zu allen in VHDL [IEEE88][IEEE93] geschriebenen Systemmodellen Fehlermodelle generieren können. Sie bestimmen anhand der Struktur einer VHDL-Komponentenbeschreibung, welche fehlerhaften Verhalten dieser Komponente möglich sein sollen (z.B. Stuck-At-If-, Stuck-At-Else-Fehler).

Ein so gewonnenes, einfaches Fehlermodell läßt sich durch eine Steuerdatei für den Fehlersimulator, die Informationen über die Auftrittswahrscheinlichkeiten der einzelnen Fehler und die Verteilungen ihrer Auftrittsdauer enthält, zu einem vollständigen Fehlermodell ergänzen.

Schwierigkeiten ergeben sich bei diesem Ansatz einerseits durch das Fehlen der Fehlerraten und der Verteilungen der Fehlerdauern (siehe Abschnitt 2.2). Diese sind gegebenenfalls für jede generierte Fehlermöglichkeit in die Simulator-Steuerdatei nachzutragen.

Ein weiteres Problem ist die Aufteilung des Fehlermodells auf mehrere, nicht direkt zusammenhängende Beschreibungen (Modell, Simulator und Simulator-Steuerdatei). Durch diese Trennung kann es leicht zu Inkonsistenzen zwischen den einzelnen Modellteilen kommen, wodurch gegebenenfalls falsche Simulationsergebnisse gewonnen werden. Bei größeren Systemen sind Inkonsistenzen in der Praxis nicht zu vermeiden, da die Anzahl der möglichen Fehler bei großen Systemen unübersehbar wird (siehe Abschnitt 2.4).

Andererseits ergibt sich durch die Generierung des Fehlermodells aus einer Systembeschreibung auf z.B. System- oder Register-Transfer-Ebene das Problem, daß die auf hoher Ebene beschriebenen Fehler i.a. nicht mit Fehlern in der realen Hardware korrespondieren (siehe Abschnitt 2.5).

### 2.1.2 Instrumentierte Systeme

Alle instrumentierten Systeme verwenden die bestehende, reale Hard- und Software sowie die reale Umgebung als Grundlage. Das reale System wird als sein eigenes Systemmodell verwendet. Alle zusätzlichen Hardware- und/oder Software-Anteile dienen dazu, das Fehlermodell zu implementieren und zusätzliche Beobachtungsmöglichkeiten zu schaffen. Durch die Beschreibung der Instrumentierung wird das Fehlermodell dargestellt.

Der generelle Vorteil dieses Verfahrens ist, daß in gleicher Zeit sehr viel mehr Experimente durchgeführt werden können als mit einer reinen Simulation. Als Nachteile sind die beschränktere Beobachtbarkeit und die erhöhten Kosten durch die Instrumentierung zu nennen. Zudem ist dieses Verfahren noch nicht während der Design-Phase eines Systems anwendbar, da zumindest ein Prototyp des Systems vorhanden sein muß.

Im folgenden werden die Fehlermodelle sowie die Vor- und Nachteile verschiedener, bestehender Verfahren erläutert.

#### Hardware-instrumentierte Systeme

Die Verfahren mit Hardware-Instrumentierung besitzen gegenüber den Software-Instrumentierungsverfahren den Vorteil, daß sie den normalen, fehlerfreien Ablauf des Systems nicht beeinflussen. Hardware-Lösungen können jedoch z.B. bei zu untersuchenden Multiprozessoren große zusätzliche Hardware-Kosten verursachen.

In **FTMP** [Lala83], **MESSALINE** [Arlat90], **FOCUS** [Choi92], **HYBRID** [Young92b], **RIFLE** [Madeira94] und **SCRIBO** [Steininger97] werden durch kleine Kontakte zusätzliche Ladungen auf die Leiterbahnen des vorhandenen Systems aufgebracht bzw. die Pins einzelner Chips auf einen bestimmten Pegel gelegt. D.h. das Fehlermodell beschreibt nur kurzzeitige (FOCUS) oder sowohl temporäre als auch permanente (FTMP, MESSALINE, HYBRID, RIFLE

## 2. Modellaufbau

---

und SCRIBO) Pegeländerungen an einigen Ein- oder Ausgängen von Subkomponenten des Systems. Der Typ des Fehlers (positive/negative Ladung bzw. Stuck-At-0/1), der Ort, der Zeitpunkt und die Dauer des Auftretens eines Fehlers werden vom Benutzer selbst über Steuerdateien angegeben.

Ein Vorteil dieses Verfahrens ist, daß die Experimente sehr gut steuerbar und beobachtbar sind. Die Fehler können zeitlich auf Clock-Zyklen genau injiziert und ausgewertet werden.

Da die Anzahl der Injektionsorte auf eine kleine Zahl beschränkt bleiben muß, ist fraglich, wieviel Prozent der in der Realität auftretenden Fehler mit Hilfe der Instrumentierung injiziert werden können (siehe Abschnitt 2.5 und Kapitel 6). Es können nur relativ wenige Signale gestört werden, da der Platz in einem System häufig nicht ausreicht, um viele Klemmen o.ä. anzubringen und viele Signale nicht zugänglich sind (z.B. Chip-interne Signale). Weiterhin ist eine Ansteuerschaltung für Änderungsmöglichkeiten an vielen Signalen aufwendig.

[Cusick86], [Karlsson89] und [Gunnflo89] beschreiben, wie durch Schwerionen-Beschuß Störungen auf das zu testende System übertragen werden können. In [Gunnflo90] wird erläutert, wie mit Hilfe von Versorgungsspannungsschwankungen Fehler in Computer-Systeme injiziert werden können.

Der Vorteil dieser Verfahren ist, daß sie Fehler realitätsnah beschreiben. Störungen, die am realen System auftreten können, werden durch gleiche Einflüsse wie in der Realität erzeugt.

Problematisch ist jedoch, daß sich zwar bei den genannten Verfahren die Größe und die Dauer der Störung einstellen läßt, der genaue Auftrittsort eines Fehlers jedoch nicht festlegbar ist. Der Fehler tritt „irgendwo“ im betroffenen Bereich auf. Unter Umständen werden durch eine einzelne Störung auch mehrere Fehler auf einmal provoziert. Es ist selbst nach einem Experiment nicht feststellbar, an welchen Orten wirklich Fehler aufgetreten sind. Unter Umständen treten im Verhältnis sehr viel häufiger Mehrfachfehler auf, die in der Realität - bedingt durch eine sehr viel kleinere Fehlerrate - praktisch nicht vorkommen (siehe Abschnitt 3.2).

### Software-instrumentierte Systeme

Eine große Zahl von Ansätzen versucht, bestehende Systeme mit Hilfe von Software-Erweiterungen bezüglich Fehlertoleranzeigenschaften zu bewerten. Dazu wird auf den entsprechenden Systemen parallel zur Nutzenanwendung ein zusätzliches Programm (ein sogenannter Fehlerinjektor) geladen, welches das einfache Fehlermodell des Systems beschreibt. Eine Steuerdatei für den Fehlerinjektor enthält die Zeitinformationen für das vollständige Fehlermodell. Zu den Ansätzen, die auf diese Weise das Modell eines Systems beschreiben, gehören **FERRARI** [Kanawati92], **EFA** [Echtle92], **ProFI** [Lovric95], **SFI** [Rosenberg93], **CSFI** [Carreira95a], **FINE** [Kao93], **FIAT** [Segall88][Barton90], **DOCTOR** [Han93], **Xception** [Carreira95b], sowie [Sieh93] und [Sieh94].

Ein Problem dieses Ansatzes ist, daß die einzelnen Teile des Modells getrennt in völlig verschiedenen Darstellungsformen vorliegen und daher eine Konsistenz des Gesamtmodells nur schwer gewährleistet werden kann (siehe Abschnitt 2.4).

In den vorliegenden Implementierungen sind zudem die Fehlermodelle sehr einfach gehalten. So sind meist nur Fehler auf Register-Transfer-Ebene (z.B. Bit-Fehler, Übertragungsfehler auf internen und externen Bussen) oder Systemebene (z.B. Nachrichtenverlust und - verfälschung) implementiert. Da einige Hardware-Bereiche für direkte Software-Zugriffe nicht sichtbar (z.B. Cache) oder nicht zugänglich (z.B. Befehls-Pipeline der CPU) sind, gestaltet sich die Fehlermodellimplementierung zum Teil sehr aufwendig. Im Extremfall muß die Hardware nahezu voll-

ständig simuliert werden, um Fehler in bestimmten Komponenten nachbilden zu können. Die Fehlerinjektion wird sich daher immer auf Fehler auf höherer Ebene (z.B. Register-Transfer- oder Systemebene) beschränken (siehe Abschnitt 2.5).

Schließlich verfälscht der geladene Fehlerinjektor das Hardware-Modell durch einen eigenen Ressourcenverbrauch (z.B. Speicher und Rechenzeit) und direkte Systemänderungen. So verändern z.B. die meisten Fehlerinjektoren den Timer-Interrupt, um nach bestimmten Zeiten Fehler zu injizieren.

## 2.2 Fehlermodell mit Zeitparametern

In den meisten Veröffentlichungen werden die Fehler eines Systems, über mehrere verschiedene Beschreibungen verteilt, dem Analysewerkzeug mitgeteilt. Die möglichen Fehler und ihre Auswirkungen sind im Fehlerinjektor beschrieben. Die Angabe, wann und für wie lange die einzelnen Fehler injiziert werden, ist Teil der Steuerdatei für den Fehlerinjektor. Werden die Fähigkeiten des Fehlerinjektors oder der Inhalt der Steuerdateien nicht exakt beschrieben, sind derartige Experimente von anderen Personen nicht nachvollziehbar.

Zu einem vollständigen Fehlermodell gehören neben der Beschreibung, welche Fehler auftreten können, auch die Zeitparameter für diese Fehler: wie häufig sie im Mittel auftreten und welche zeitliche Verteilung für ihre Dauer anzunehmen ist.

Außer in REACT, DEPEND und SimPar fehlen in den Modellen die Zeitparameter für das Fehlermodell vollständig. Die genannten Werkzeuge unterstützen die Angabe einer Fehlerrate. Die Fehlerauftrittsverteiling ist i.a. exponentiell verteilt, kann bei DEPEND und SimPar jedoch vom Benutzer umdefiniert werden. Die Dauer eines Fehlers wird indirekt über eine zusätzliche Reparaturkomponente angegeben. Die Reparaturdauer entspricht bei dieser Art der Modellierung der Dauer des Fehlers. Für Modelle auf Systemebene erscheint dies sinnvoll. Zur Modellierung von temporären Fehlern ist eine Angabe der Verteilungsfunktion für die Fehlerdauer bei der zugehörigen Fehlerdefinition weitaus übersichtlicher.

Wenn nur ein einfaches Fehlermodell gegeben ist, d.h. die Angaben über die zeitlichen Parameter der einzelnen Fehler nicht vorhanden sind, kann ein Modell nicht bezüglich Fehlertoleranzeigenschaften analysiert werden. Ein Benutzer, der ein solches Modell zur Analyse verwenden will, muß selbst (z.B. in der Steuerdatei für den Fehlerinjektor) diese Daten nachtragen. Dazu ist jedoch eine genaue Kenntnis des gesamten Modells bis in das kleinste Detail notwendig. Wird das Modellverhalten bezüglich einer bestimmten Gruppe von Fehlern nicht getestet (z.B. da diese nicht bekannt waren), kann dies unter Umständen große Auswirkungen auf die erhaltenen Resultate haben. Da aber jede kleine Änderung des Systemmodells eine Änderung der möglichen Fehler und ihrer Zeitparameter nach sich zieht, ist eine vollständige Kenntnis aller möglichen Fehler im System von einer einzelnen Person bei großen Systemen praktisch nicht zu erreichen. Die Resultate werden daher immer ungenau sein.

Am Beispiel der Ausfallrate  $\mu$  wird deutlich, wie sehr die erhaltenen Resultate von der Auftrettsrate der einzelnen Fehler  $\lambda_f$  und der Menge der möglichen Fehler  $F$  abhängt ( $p_f$ : Wahrscheinlichkeit, daß das System bedingt durch den Fehler  $f$  ausfällt) [Arlat90]:

$$\mu = \sum_{f \in F} \lambda_f p_f$$

Ohne Kenntnis der Parameter  $F$  und  $\lambda_f$  ist  $\mu$  nicht einmal abschätzbar.

Häufig wird argumentiert, daß die Parameter  $F$  und  $\lambda_f$  nicht bekannt sind und daher auch kein passendes Modell aufgestellt werden kann. Im Normalfall wird jedoch ein Modell während der Design-Phase des Systems aufgestellt. Die Fehlerraten werden also vom Designer der Hardware vorgegeben und sind als Design-Entscheidungen (wie z.B. maximale Verzögerungszeiten von Gattern auch) anzusehen, die vom Hardware-Hersteller einzuhalten sind.

Auch im Falle, daß vorhandene Hardware mit Hilfe eines Modells nachgebildet und bezüglich Fehlertoleranzeigenschaften bewertet werden soll, ist die Angabe der Fehlerraten im Modell hilfreich. Durch die Angabe der Raten ist eine Dokumentation der Annahmen gegeben, die zu den gewonnenen Ergebnissen der Bewertung des Systems führen.

### 2.3 Trennung von Modell und Modellierungswerkzeug

In der Design-Phase eines Systems ist es sehr häufig erforderlich, das Modell zu verändern, um z.B. die bei der Analyse oder bei einer Simulation erkannten Schwächen des Systems zu beheben. Ein gutes Modellierungswerkzeug sollte diese Änderungen am Modell möglichst wenig behindern und nicht in bestimmte Richtungen beeinflussen. Zudem soll ein Modellierungswerkzeug i.a. nicht nur zum Design eines einzigen Systems herangezogen werden. Ein gutes Modellierungswerkzeug zeichnet sich im Gegenteil dadurch aus, daß es Hilfestellung bei der Beschreibung und Analyse möglichst vieler Modelle geben kann. Daher sollte eine strikte Trennung von Modell und Auswertewerkzeug existieren. Ist diese Trennung nicht gegeben, wird durch die Verwendung eines bestimmten Modellierungswerkzeugs ein Teil des Modells (z.B. das Fehlermodell) unveränderbar vorgegeben.

Dies ist ein großes Problem bei den Hardware-basierten Fehlerinjektoren (z.B. Pin-Level-Fehlerinjektoren, Schwerionen-Strahlung). Die Methode der Fehlerinjektion gibt in diesen Fällen das Fehlermodell vor. So ist es z.B. nicht möglich, bei einem Pin-Level-Fehlerinjektor das Fehlermodell um Prozessor-interne Bit-Flip-Fehler zu erweitern oder bei einem Fehlerinjektor mit Schwerionen-Strahlung Pin-Level-Fehler einzuführen. Auch kann die Hardware des zu untersuchenden Systems nicht beliebig verändert werden, ohne den Fehlerinjektor anzupassen. So kann beispielsweise eine Erhöhung der Taktrate oder Veränderung der Versorgungsspannung des Systems eine völlige Neuentwicklung eines Pin-Level-Fehlerinjektors erfordern. Hardware-basierte Fehlerinjektoren sind daher immer nur für ein bestimmtes, unveränderbares Fehlermodell und eine nur in Grenzen modifizierbare Hardware geeignet.

Ähnliches gilt für Software-implementierte Fehlerinjektoren. Ein unveränderbarer Bestandteil aller bekannten Injektoren sind die Regeln, nach denen der Zustand des modellierten Systems während der Laufzeit modifiziert werden kann. Das Fehlermodell kann daher nicht vom Modellierer, sondern nur vom Hersteller des Fehlerinjektors geändert werden. Da jede Hardware-Änderung auch eine Fehlermodelländerung verlangt, sind diese Fehlerinjektoren auch nur für eine ganz bestimmte Hardware geeignet oder verwenden ein entsprechend ungenaues Fehlermodell.

Ein ähnliches Problem besteht bei den Modellierungswerkzeugen, die bestehende Modelle gemäß bestimmten Regeln nachträglich automatisch um Fehlermodelle erweitern (z.B. **MEFI-STO** [Jenn94]). Zwar kann in diesem Fall das Systemmodell ohne Aufwand geändert werden (das Fehlermodell wird automatisch angepaßt), das Fehlermodell kann jedoch nicht individuell modifiziert werden. Die Änderung des Fehlermodells ist unmöglich, da die Regeln, gemäß denen ein Systemmodell um Fehler erweitert wird, im Werkzeug verborgen sind, das automatisch ein passendes Fehlermodell zu einem gegebenen Hardware-Modell erzeugt. Eine Modelländerung bedingt in diesem Fall auch eine Werkzeugänderung.

Es sind jedoch eine Vielzahl verschiedener Beschreibungsebenen sowohl für das System-als auch für das Fehlermodell denkbar. Als Beispiele seien hier genannt ([Jacomet91], [Khare95]):

**Tabelle 1: Verschiedene Systemmodellierungsebenen und ihre Fehlermodelle**

System-Beschreibungsebene	Fehlermodell
Verhaltensebene	Pin-Level-Fehler fehlende Nachrichten zusätzliche Nachrichten verfälschte Nachrichten
Register-Transfer-Ebene	Busfehler Register-Bit-Fehler fehlerhafte Operationen
Gatterebene	Stuck-At-Fehler Stuck-Open-Fehler Stuck-Toggle-Fehler Logic-Large-Scope-Short Current-Large-Scope-Short
Schaltungsebene	Kurzschluß Unterbrechung fehlerhafte Ladungen fehlerhafte Halbleiterfunktionen
Layout-Ebene	zusätzliches Material fehlendes Material verändertes Material Leitungsübersprechen

*Ziel bei der Entwicklung einer Modellierungsumgebung für die Untersuchung von Fehlertoleranzeigenschaften sollte daher eine strikte Trennung des zu untersuchenden Modells von der verwendeten Modellierungsumgebung sein. Das heißt, daß sowohl das Systemmodell als auch das Fehlermodell eine vom Modellierer und nicht vom Modellierungswerkzeug-Hersteller aufzustellende Beschreibung sein sollte.*

## 2.4 Verschmelzung von System- und Fehlermodell

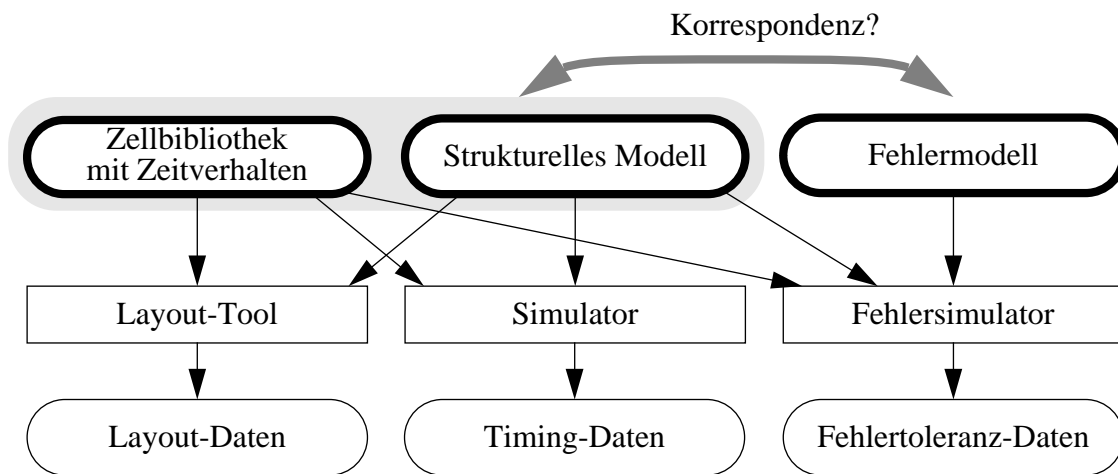
Außer in Modellierungsumgebungen wie REACT, DEPEND und SimPar sind System- und Fehlermodell voneinander unabhängige Beschreibungen. In anderen Modellierungsumgebungen werden Systeme durch die reale Hard- und Software sowie die reale Umgebung bzw. durch ein entsprechendes Systemmodell (z.B. VHDL-Modell) beschrieben, während die möglichen Fehler mit ihren Auswirkungen, Raten usw. innerhalb des Fehlerinjektors bzw. Fehlersimulators mit den entsprechenden Steuerdateien bestimmt werden.

Diese Trennung der Beschreibungen hat den Vorteil, daß bereits vorhandene Systemmodelle, die z.B. während der Design-Phase (ohne Berücksichtigung von Fehlertoleranzaspekten) oder zur Leistungsanalyse erstellt wurden, weiterverwendet werden können. Es wird nur ein Teil, das Fehlermodell, zum ursprünglichen Modell hinzugefügt.

## 2. Modellaufbau

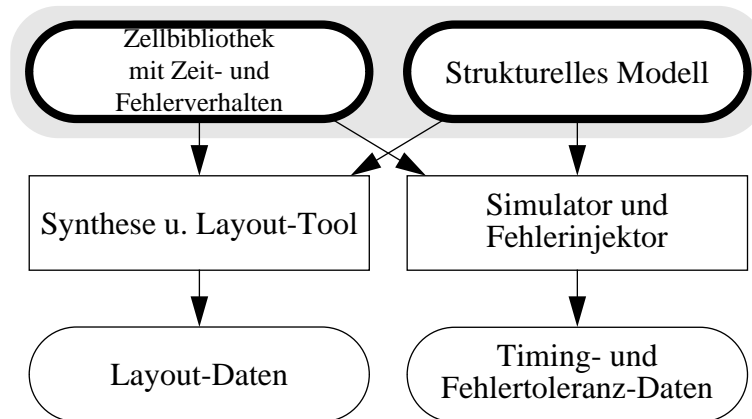
Der große Nachteil dieser Aufteilung ist jedoch, daß eine Korrespondenz zwischen dem Systemmodell und dem Fehlermodell nur schwer herzustellen ist. Wird die Hardware auch nur minimal geändert, muß auch das Fehlermodell entsprechend angepaßt werden. Unterbleibt die Anpassung, sind möglicherweise die erhaltenen Analyseergebnisse falsch. Sind die zu untersuchenden Systeme entsprechend groß (mehrere Millionen Gatter), ist die Korrespondenz der beiden Teilmodelle praktisch nicht mehr herzustellen (siehe Abbildung 1). Dies gilt insbesondere dann, wenn das Systemmodell nicht vom Modellierer selbst geschrieben wird, sondern durch ein Synthese-Tool aus einer abstrakteren Beschreibung automatisch generiert wird (siehe Abschnitt 2.5).

In den folgenden Diagrammen werden alle Datenbestände durch Kästchen mit abgerundeten Ecken dargestellt, während die einzelnen Werkzeuge durch Rechtecke repräsentiert werden. Die Eingabedaten des Benutzers sind zusätzlich dick umrandet und alle Daten, die das Systemmodell beschreiben, sind grau hinterlegt. Die Pfeile deuten den Datenfluß an. Sie zeigen für jedes Werkzeug, welche Daten es verwendet und welche es erzeugt.



**Abb. 1: Generelles Korrespondenzproblem von System- und Fehlermodell**

Eine bessere Alternative wird durch Abbildung 2 dargestellt. Hier sind das System- und das Fehlermodell miteinander verschmolzen. Jede Komponente des Systems enthält sowohl eine Beschreibung ihres normalen Verhaltens als auch ihrer möglichen Fehler. Innerhalb einzelner Komponenten ist die Korrespondenz zwischen ihrem Systemmodell und den dazugehörigen Fehlern i.a. leicht zu gewährleisten, da einzelne Komponenten verhältnismäßig klein und entsprechend übersichtlich sind. Wird ein großes Modell aus vielen derartigen Komponenten „zusammengesteckt“, ist immer eine Korrespondenz von System- und Fehlermodell gegeben, da beim Hinzufügen (Austauschen, Entfernen) von Komponenten zum Modell automatisch die Beschreibungen ihrer möglichen Fehler mit hinzugefügt (ausgetauscht, entfernt) werden.



**Abb. 2: Verschmolzenes System- und Fehlermodell**

Ein Nachteil dieses Verfahrens ist, daß eine neue Modellierungssprache notwendig ist. Keine der derzeit auf dem Markt befindlichen Sprachen erlaubt eine effiziente Hardware-Modellierung und gleichzeitig eine stochastische Modellierung innerhalb eines einzigen (Teil-) Modells. Abschnitt 2.6 zeigt jedoch am Beispiel der Modellierungssprache VHDL, wie Hardware-Modellierungssprachen auf einfache Weise um stochastische Elemente erweitert werden können.

## 2.5 Fehlermodelle auf niedriger Abstraktionsebene

Beim Design-Prozeß werden im allgemeinen Verhaltensmodelle oder Modelle auf Register-Transfer-Ebene erstellt, die von handelsüblichen Synthese-Werkzeugen in mehreren Schritten automatisch zu Modellen auf Gatter-, Schaltungs- und Layout-Ebene transformiert werden. Dieses Vorgehen hat den Vorteil, daß die Komponenten auf einer Ebene beschrieben sind, die für den Menschen übersichtlich und damit verständlich sind. Dabei stellt sich die Frage, auf welcher Ebene Fehler modelliert werden sollten. Folgende Möglichkeiten bestehen: Fehler werden beschrieben auf

- Verhaltensebene,
- Register-Transfer-Ebene
- Gatterebene
- Schaltungsebene

oder

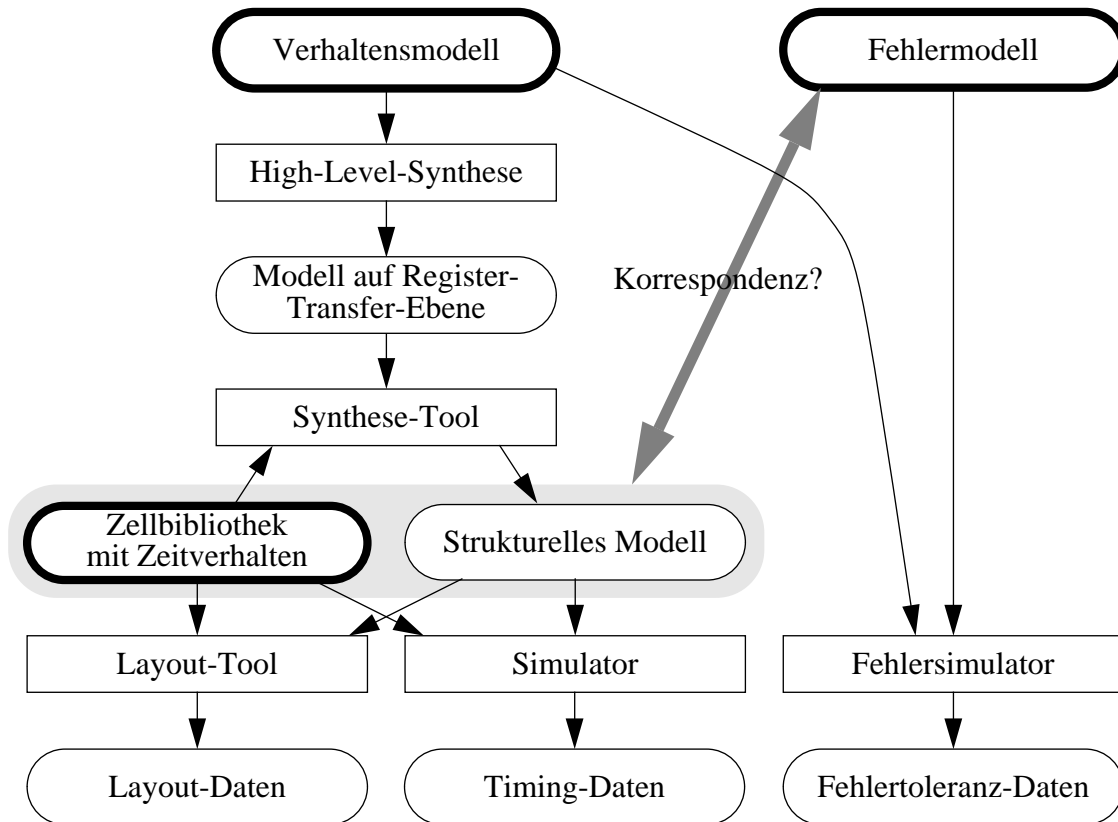
- Layout-Ebene.

Jede dieser Möglichkeiten hat ihre Vor- und Nachteile, die im folgenden beschrieben werden sollen.

Der Vorteil der Fehlerbeschreibung auf Verhaltensebene liegt in der effizienten Auswertbarkeit derartiger Modelle. Sind Komponenten auf Verhaltensebene modelliert, existieren im Vergleich zu Modellen auf niedrigerer Ebene sehr viel weniger einzelne Prozesse und Signale (siehe Abschnitt 3.7), was eine schnellere Simulierbarkeit bedeutet.

## 2. Modellaufbau

Abbildung 3 zeigt jedoch ein gravierendes Problem der Fehlerbeschreibung auf Verhaltensebene. Da durch verschiedenen Synthese-Schritte aus dem Verhaltensmodell ein Modell auf Gatterebene generiert wird, ist die Korrespondenz von Gattermodell zu Fehlermodell nicht zu gewährleisten. Wie Kapitel 6 jedoch zeigt, sind die Analyseergebnisse eines Systems sehr sensibel bezüglich der möglichen Abbildungen des Verhaltensmodells auf eine Gatterbibliothek.

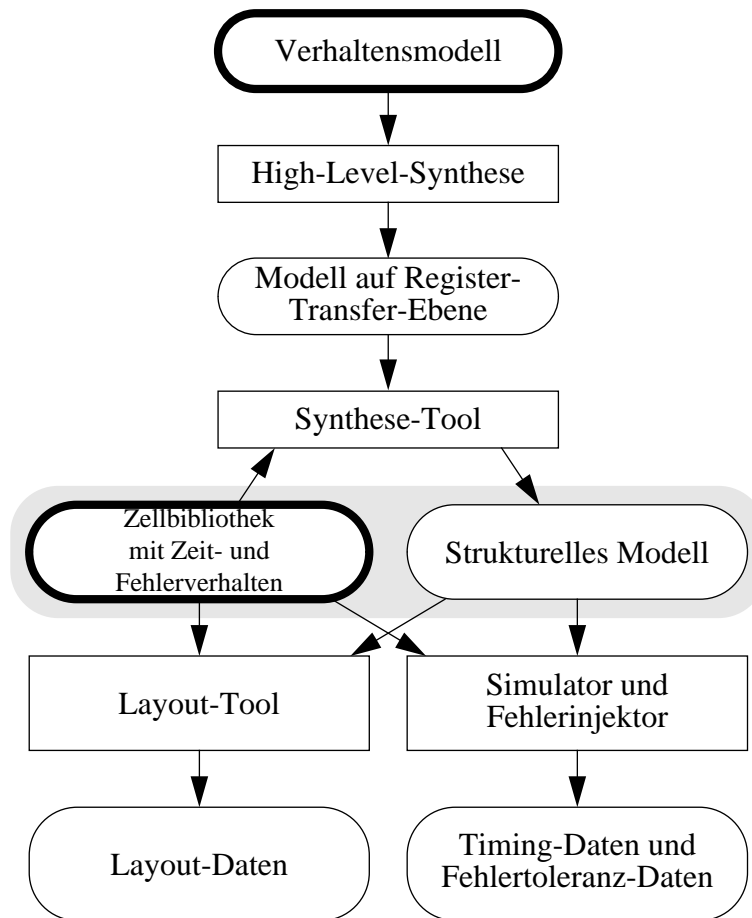


**Abb. 3: Korrespondenzproblem von System- und Fehlermodell auf Verhaltensebene**

Eine wesentlich bessere Korrespondenz der synthetisierten Hardware mit dem gegebenen Fehlermodell kann mit dem in Abbildung 4 gezeigten Verfahren erzielt werden. Das gegebene Verhaltensmodell wird mit Hilfe eines handelsüblichen Synthese-Werkzeugs auf eine Gatterbibliothek abgebildet, die das Fehlermodell enthält. Neben der besseren Korrespondenz besteht ein

## 2.5 Fehlermodelle auf niedriger Abstraktionsebene

weiterer, großer Vorteil dieser Modellierung darin, daß die Verhaltensmodelle, die für jede Hardware neu zu entwerfen sind, unverändert bleiben können. Nur die Gatterbibliothek muß ein einziges Mal um die Fehlerbeschreibungen erweitert werden.



**Abb. 4: Modellerstellung ohne durch eine Synthese eingeführte Korrespondenzprobleme**

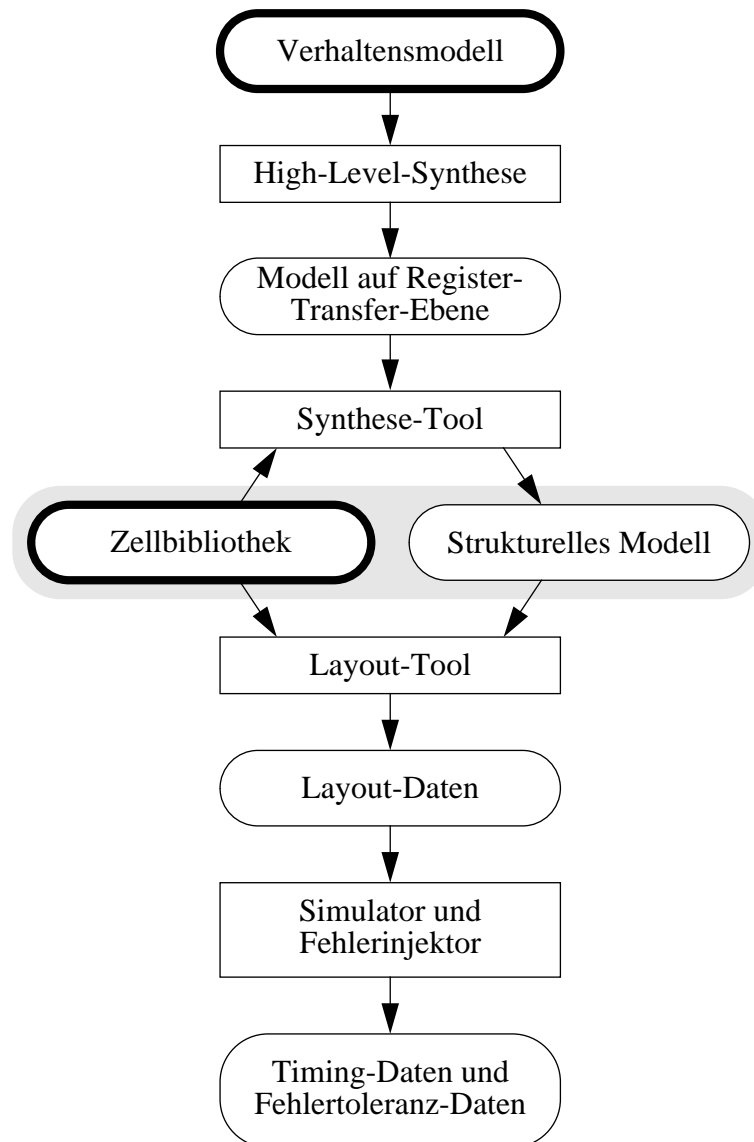
Ein Nachteil dieses Verfahrens ist, daß die Auswertung der Modelle bezüglich der Fehlertoleranzeigenschaften erst nach dem Synthese-Vorgang mit dem Modell auf Gatterebene erfolgen kann. Dies bedeutet einen erhöhten Simulations- bzw. Analyseaufwand, da gegenüber dem Verhaltensmodell deutlich mehr Prozesse und Signale vorhanden sind.

Nachteilig ist auch, daß mögliche Fehler auf Layout-Ebene nur schwer nachbildbar sind, da die Generierung der Layout-Daten erst unterhalb der Gatterebene geschieht, innerhalb derer die Fehler modelliert sind.

## 2. Modellaufbau

---

Mögliche physikalische Fehler können vom dem Ansatz, der in Abbildung 5 dargestellt ist, mit berücksichtigt werden, da die Simulation bzw. Analyse auf der Ebene der physikalischen Gegebenheiten arbeitet.



**Abb. 5: Fehlersimulation auf Layout-Ebene**

Derartige Simulationen sind zwar extrem genau, lassen sich jedoch - bedingt durch den großen Simulationsaufwand - nur für sehr kleine Modelle ausführen. Es ist undenkbar, ganze Computersysteme derartig zu modellieren und diese Modelle über größere Zeiträume zu simulieren.

Für die Praxis bedeutet dies, daß immer zwischen der Genauigkeit der Ergebnisse und dem benötigten Simulations- bzw. Analyseaufwand abgewogen werden muß (vergleiche Tabelle 1). Das Beispiel in Kapitel 6 zeigt jedoch, daß Simulations- bzw. Analyseergebnisse von auf zu hoher Ebene modellierten Systemen um Größenordnungen von den realen Werten abweichen können.

## 2.6 VHDL-Erweiterung

In den Abschnitten 2.2, 2.3, 2.4 und 2.5 wurden eine Reihe von Verbesserungen für die Modellierung von Systemen zur Bewertung von Fehlertoleranzeigenschaften vorgeschlagen. Um diese Methoden austesten zu können, wurde eine neue Modellierungssprache entwickelt, welche die entsprechenden Verbesserungsvorschläge in die Praxis umsetzt.

Um nicht von Grund auf eine neue Modellierungssprache entwickeln zu müssen und kommerzielle Synthese-Werkzeuge verwenden zu können (siehe Abschnitt 2.5), wurde eine existierende Hardware-Beschreibungssprache als Grundlage verwendet. Die Wahl fiel aufgrund bereits vorhandener Werkzeuge und der bestehenden Standardisierung auf VHDL ([Ashenden90], [IEEE88], [IEEE93]). Denkbar wäre aber auch die Verwendung anderer Hardware-Beschreibungssprachen wie z.B. HDL ([Hoffmann75]) oder VERILOG ([Golze96], [Thomas91]).

Es wurde versucht, die Erweiterungen der Sprache so klein wie möglich zu halten und die bestehende Semantik der einzelnen Sprachkonstrukte für die Ergänzungen weiterzuverwenden.

Abschnitt 2.6.1 beschreibt die eingeführten syntaktischen und semantischen Ergänzungen. Ein Beispiel für die Beschreibung von Komponenten mit dieser erweiterten Sprache zeigt der Abschnitt 2.6.2. Eine Bewertung der vorgestellten Spracherweiterung gibt Abschnitt 2.6.3.

### 2.6.1 Syntaktische und semantische Ergänzungen

In der Sprache VHDL werden die Zustände des Systems durch Werte der Signale und Variablen repräsentiert. Um zu dieser Sichtweise konsistent zu bleiben, soll auch die Existenz von Fehlern durch Werte von Signalen bzw. Variablen beschrieben werden. Da jeder Fehler unabhängig von anderen Fehlern vorhanden sein kann, ist für jeden denkbaren Fehler ein Signal bzw. eine Variable vorzusehen, die den Zustand dieses Fehlers (aktiv bzw. nicht aktiv) beschreibt.

Der Unterschied zwischen Signalen und Variablen besteht in VHDL darin, daß sich der Wert eines Signals aus der Sicht einzelner Prozesse spontan ändern kann (aufgrund von Aktivitäten anderer Prozesse), während der Inhalt von Variablen nur aufgrund von durchgeführten Wertzuweisungen des Prozesses, zu dem sie gehören, modifiziert werden kann. Geht man davon aus, daß das Auftreten eines Fehlers ein spontanes, nicht dem Prozeß zuzuordnendes Ereignis ist, müssen Fehler daher durch Signale repräsentiert werden.

Da der Zustand eines Fehlers als „aktiv“ bzw. „nicht aktiv“ angenommen wird, reicht zur Darstellung dieses Zustandes ein boolescher Wert TRUE bzw. FALSE. Für den Typ des Signals ist daher BOOLEAN sinnvoll. Denkbar wäre auch ein neuer Aufzählungstyp etwa der folgenden Art:

```
TYPE state IS ('fault_free', 'faulty');
```

Häufig wird jedoch in der Verhaltensbeschreibung – z.B. mit IF- und WHILE-Anweisungen – auf den Zustand des Systems Bezug genommen (siehe Beispiel in Abbildung 10, Zeilen 20-26). Da diese Kontrollanweisungen für die Bedingungen boolesche Werte erwarten, werden durch die Verwendung des vorhandenen Typs BOOLEAN als Signaltyp viele Typkonvertierungen eingespart.

Wie in Abschnitt 2.2 gezeigt, sollten die Parameter der einzelnen möglichen Fehler (Verteilung der Auftrittszeitpunkte und der Auftrittsdauer) Teil des Modells sein. In der entwickelten Erweiterung für die Sprache VHDL wird sowohl für die Verteilung der zeitlichen Abstände zweier Fehler als auch für die Verteilung der Fehlerdauer eine Exponentialverteilung angenommen. Im Modell sind daher zu jedem Fehler nur zwei Parameter anzugeben: der mittlere zeitliche Abstand zwischen zwei Fehlern dieses Typs sowie die mittlere Fehlerdauer. Die Angabe des mitt-

## 2. Modellaufbau

---

leren zeitlichen Abstandes (in Zeiteinheiten) ist gegenüber der inhaltlich äquivalenten Angabe der Auftrittsrates (in Anzahl pro Zeiteinheit) zu bevorzugen, da dadurch die entwickelte Sprache in sich konsistenter bleibt. Zeiteinheiten werden in VHDL an vielen Stellen verwendet (vordefinierter Typ `TIME`). Ein Typ „Rate“ müßte neu eingeführt werden.

Denkbar wäre es, für die Beschreibungen der Verteilungen von Auftrittszeitpunkt und Auftrittsdauer der einzelnen Fehler durch den Modellierer beliebige Verteilungsfunktionen angeben zu lassen. Auf diese Weise könnte z.B. die erhöhte Ausfallwahrscheinlichkeit von technischen Systemen zu Beginn und am Ende der Lebenszeit dieser Systeme mit im Modell festgehalten werden. Da es jedoch praktisch - aufgrund einer zu hohen Simulationsdauer - unmöglich ist, größere Systeme über einen derart großen Zeitraum zu simulieren, kann für den Zeitbereich, der simuliert werden soll, jeweils eine konstante Fehlerrate angenommen werden. Die zeitlichen Abstände zwischen zwei Fehlern können in diesem Fall als exponentiell verteilt angenommen werden. Es reicht also zur Beschreibung der Verteilungsfunktion die Angabe der Fehlerrate bzw. des mittleren zeitlichen Abstandes zweier Fehler.

Sind für die Angabe des mittleren zeitlichen Abstandes und der mittleren Fehlerdauer beliebige Ausdrücke (vom Typ `TIME`) erlaubt, können Fehlerraten Top-Down über den in VHDL verfügbaren Mechanismus der Generic-Parameter jeweils von übergeordneten Komponenten auf die jeweils untergeordneten aufgeteilt werden. Damit werden ganze Systeme bezüglich ihrer Fehlerraten parametrierbar.

Für die Beschreibung eines Fehlers ergibt sich daher zusammengefaßt folgendes: jeder Fehler wird charakterisiert durch ein Signal vom Typ `BOOLEAN` mit den Parametern des mittleren Abstandes zwischen zwei Fehlern (Ausdruck vom Typ `TIME`) und der mittleren Fehlerdauer (Ausdruck vom Typ `TIME`). Dies läßt sich auf einfache Weise in die Grammatik der Sprache VHDL einbetten. Überall, wo in VHDL Signale erlaubt sind, sollen auch Fehlersignale erlaubt sein. D.h. die Grammatikregel<sup>1</sup> `signal_declaration` ist wie folgt zu ändern (EBNF):

```
signal_declaration ::=
    SIGNAL identifier_list : subtype_indication
    [signal_kind] [:= expression];
```

**Abb. 6: Regel für Signal-Deklaration in VHDL-93**

`identifier_list` ist hier eine durch Kommata getrennte Liste von Namen, welche die mit dieser Deklaration instanziierten Signale bekommen sollen. Der Datentyp der Werte, die durch die deklarierten Signale übertragen werden können, werden durch die Regel `subtype_indication` bestimmt. Sind die Signale Elemente eines Busses, d.h. können mehrere Prozesse auf die Signale schreibend zugreifen, kann über die Angabe `signal_kind` optional angegeben werden, wie sich das Signal verhalten soll, wenn zu einem bestimmten Zeitpunkt kein Prozeß das Signal treibt. Durch eine mögliche `expression` kann den Signalen ein initialer Wert zugewiesen werden.

---

1. Es wird im folgenden die Grammatik aus [IEEE93] verwendet.

```

signal_declaration ::=
    SIGNAL identifier_list : subtype_indication
    [signal_kind] [:= expression];
|
    SIGNAL identifier_list : subtype_indication
    INTERVAL expression DURATION expression;

```

**Abb. 7: Regel für Signal-Deklaration im erweiterten VHDL**

Mit dieser Erweiterung ist es möglich, spontane Zustandsänderungen eines System aufgrund von Fehlern eindeutig zu beschreiben. Auf diese zusätzlichen Zustandsvariablen des Systems kann im darauffolgenden Abschnitt, der das Verhalten der Komponente beschreibt, Bezug genommen werden. Dadurch ist es möglich, nicht nur die Fehler an sich, sondern auch ihre Auswirkungen auf das Verhalten der Komponente mit in die Beschreibung der einzelnen Komponenten aufzunehmen. Somit kann auch der Effekt der Fehler eindeutig modelliert werden.

## 2.6.2 Beispiel

Im folgenden Beispiel soll gezeigt werden, wie diese Erweiterung in der Praxis verwendet werden kann. Modelliert ist hier ein einfaches NOT-Gatter mit einem Stuck-At-Fehlermodell.

```

0: LIBRARY ieee;
1: USE ieee.std_logic_1164.ALL;
2:
3: ENTITY not_gate IS
4:     PORT( i: IN      std_logic;
5:           o: OUT    std_logic);
6: END not_gate;

```

**Abb. 8: NOT-Gatter: Entity-Beschreibung**

```

7: ARCHITECTURE behavior OF not_gate IS
8: BEGIN
9:     PROCESS(i)
10:    BEGIN
11:        o <= NOT i AFTER 1 ns;
12:    END PROCESS;
13: END behavior;

```

**Abb. 9: NOT-Gatter: Verhaltensmodell ohne Fehlermodell**

## 2. Modellaufbau

---

```
14: ARCHITECTURE faulty_behavior OF not_gate IS
15:     SIGNAL sa0: boolean INTERVAL 15000 hr DURATION 6 us;
16:     SIGNAL sa1: boolean INTERVAL 60000 hr DURATION 2 us;
17: BEGIN
18:     PROCESS(i, sa0, sa1)
19:     BEGIN
20:         IF sa0 THEN
21:             o <= '0';
22:         ELSIF sa1 THEN
23:             o <= '1';
24:         ELSE
25:             o <= NOT i AFTER 1 ns;
26:         ENDIF;
27:     END PROCESS;
28: END faulty_behavior;
```

**Abb. 10: NOT-Gatter: Verhaltensmodell mit integriertem Fehlermodell**

Die Entity-Beschreibung (Abbildung 8) ändert sich bei Einführung eines Fehlermodells für die zu modellierende Komponente nicht. Dies ist auch sinnvoll, da Fehler als Komponenten-intern anzusehen sind. Werden Fehler von außen auf eine Komponente übertragen (z.B. Störungen der Stromversorgung) kann dies durch zusätzlich eingeführte Ports in der Entity-Deklaration modelliert werden (z.B. Ports für Vcc und GND zur Modellierung von Stromversorgungsstörungen).

Die Code-Fragmente in den Abbildungen 9 und 10 zeigen Modelle für das NOT-Gatter ohne Fehlermodell (Abbildung 9) und mit integriertem Fehlermodell (Abbildung 10). Zur Integration der Fehlerbeschreibung sind in den Zeilen 15 und 16 zwei Fehlersignale eingefügt. Sie beschreiben zwei Fehler (*sa0* bzw. *sa1*), die mit mittleren Raten von einem Fehler pro 15000 Stunden bzw. pro 60000 Stunden auftreten. Für ihre Dauer ist im Mittel ein Wert von 6 Mikrosekunden (Fehler *sa0*) bzw. 2 Mikrosekunden (Fehler *sa1*) anzunehmen. Wie Modelle mit derartigen Unterschieden in den Größenordnungen der einzelnen Zeitangaben (hier z.B. 60000 Stunden bzw. 2 Mikrosekunden) effizient ausgewertet werden können, ist in Kapitel 3 („Effiziente Experimentdurchführung“) beschrieben.

Da das Verhalten des Gatters von den Fehlern abhängig ist, müssen die dazugehörigen Fehler der Sensitivity-List in Zeile 18 hinzugefügt werden. So wird der Prozeß bei jedem Wechsel des Zustandes der Komponente (fehlerfrei -> fehlerhaft bzw. fehlerhaft -> fehlerfrei) aktiviert und kann die Ausgabewerte entsprechend anpassen. Wie der Prozeß auf die genannten Fehler reagiert, wird in den Zeilen 20 bis 26 beschrieben. Die Zeilen besagen, daß, wenn der Fehler *sa0* aktiv ist, am Ausgang der Komponente eine '0' erscheint (Zeile 20 und 21), wenn der Fehler *sa1* aktiv ist, der Ausgabe-Port *o* eine '1' zugewiesen bekommt (Zeile 22 und 23). Nur in dem Fall, daß kein Fehler aktiv ist, ist das normale Verhalten der Komponente zu erwarten (Zeile 24 und 25).

### 2.6.3 Bewertung

Wie aus der Beschreibung der syntaktischen und semantischen Erweiterungen in Abschnitt 2.6.1 zu sehen ist, beschränken sich die Änderungen auf einen sehr kleinen Bereich der Sprache VHDL. Für Personen, die bereits an den Umgang mit Standard-VHDL gewöhnt sind, ist es deshalb sehr schnell möglich, diese Änderungen der Sprache VHDL zu erlernen. Verschiedene

Personen haben diese erweiterte Sprache verwendet. Dabei traten keinerlei Verständnisprobleme bezüglich der Erweiterungen - weder syntaktischer noch semantischer Art - auf. Im allgemeinen reichen kleine Beispiele (wie z.B. Abbildung 10), um die Bedeutung der neuen Erweiterungen zu verdeutlichen.

Wie Erfahrungen gezeigt haben ([Bogendörfer96], [Sieh97a], [Sieh97b], [Sieh97c], [Sieh97d], [Stiborsky97]), konnten VHDL-Modelle mit dieser neuen Sprache leicht um ihr zugehöriges Fehlerverhalten erweitert bzw. von Grund auf mit Fehlerbeschreibungen neu erstellt werden. Mit Ausnahme der Arbeit von [Bogendörfer96] wurden Modelle nach der in Abbildung 4 dargestellten Methode aufgebaut, die sich in der Praxis sehr bewährt hat. Die Generierung der Modelle ist damit – trotz der erweiterten Funktionalität – nicht komplizierter und auch nicht umfangreicher als der Aufbau von VHDL-Modellen ohne Fehlerbeschreibung. Lediglich ein einziges Mal mußte Arbeit in die Zellbibliothek investiert werden, um die Fehlerbeschreibungen für die einzelnen Zellen zu integrieren. Diese Arbeit konnte jedoch in wenigen Tagen durchgeführt werden.



# 3. Effiziente Experimentdurchführung

Nach der Aufstellung eines Modells sollen Fehlertoleranzmaße (Ausfallrate, Verfügbarkeit, Zuverlässigkeit o.ä.) des Systems ermittelt werden. Einfache Modelle können analytisch, kompliziertere nur noch simulativ ausgewertet werden.

Analytische Verfahren sollen an dieser Stelle nicht betrachtet werden, da sie nur für relativ einfache Modelle mit wenigen Komponenten anwendbar sind. Gerade große Systeme, für die Fehlertoleranzmaßnahmen notwendig sind, bestehen jedoch aus einer so großen Anzahl von Einzelkomponenten, daß eine analytische Auswertung undenkbar ist. Die Verwendung eines abstrakteren Modells mit weniger Komponenten, das das gleiche System auf höherer Ebene modelliert, ist nach den Ergebnissen aus Kapitel 6 nicht zu empfehlen, da die damit gewonnenen Ergebnisse i.a. zu ungenau sind.

Sowohl die analytischen Auswertungen als auch die Simulationsverfahren sind sehr rechenintensiv. Ihre Ausführung sollte daher möglichst effizient sein [Iyer93]. Auch in Echtzeit ablaufende Simulationsläufe sollten - wenn möglich - noch beschleunigt werden, da für eine ausreichende statistische Genauigkeit meist eine sehr große Anzahl von Fehlerszenarien getestet werden muß. Wenn möglich, soll ein Verfahren, Experimente zu beschleunigen, so allgemein sein, daß es für alle denkbaren, zu simulierenden Systeme anwendbar ist. Dabei ist jedoch zu beachten, daß auf keinen Fall das Ergebnis des Experiments verfälscht wird.

Folgende Phasen werden bei einer Fehlersimulation durchlaufen:

- eine fehlerlose Phase
- eine Phase, in der Fehler aktiviert (injiziert) werden
- eine Überprüfungsphase, um die Auswirkungen der Fehlerinjektion zu protokollieren

Jede dieser einzelnen Phasen sollte so effizient wie möglich simuliert werden können. Dazu läßt sich die Tatsache ausnutzen, daß mehrere, in weiten Bereichen gleichablaufende Experimente durchgeführt werden. Beispielsweise ist jeweils der Ablauf der Simulation vor der Fehlerinjektion gleich.

Wenn möglich, sollten Fehler daher so injiziert werden, daß eine vom Benutzer vorgegebene Konfidenz der Ergebnisse mit einer möglichst kleinen Anzahl von Injektionen erreicht werden kann. Hierfür lassen sich Informationen ausnutzen, die a priori bekannt sind (z.B. Fehlerraten). So kann es beispielsweise sinnvoll sein, die Auswirkungen von häufiger auftretenden Fehlern genauer zu bestimmen als die Auswirkungen von Fehlern mit kleinerer Fehlerrate.

Nach einem Überblick über bestehende Simulationsverfahren in Abschnitt 3.1 werden in den darauffolgenden Abschnitten 3.2 bis 3.9 eine Reihe von Methoden entwickelt und bezüglich ihrer Wirksamkeit zur Beschleunigung der Simulation von Fehlerinjektionsexperimenten überprüft.

## 3.1 Bestehende Simulationsverfahren

Die zur Zeit verwendeten Simulationsverfahren (Übersicht: [Clark95], [Iyer94]) unterscheiden sich zum Teil erheblich. Abhängig vom benutzten Modellierungsverfahren (virtuelle Systeme bzw. instrumentierte Systeme, siehe Abschnitt 2.1) werden grundsätzlich verschiedene Verfahren eingesetzt. Zur Simulation von virtuellen Systemen werden ihre Beschreibungen durch

### 3. Effiziente Experimentdurchführung

---

Compiler in lauffähige Programme umgewandelt, die während ihres Laufes das Verhalten des Systems nachbilden. Instrumentierte Systeme „simulieren“ sich selbst, während ihre Instrumentierungen zur Fehlerinjektion Veränderungen an den Systemen durchführen.

Der prinzipielle Ablauf ist in allen in Abschnitt 2.1 erwähnten Modellierungswerkzeugen gleich. Die Simulation wird für eine bestimmte Zeit ausgeführt. Zu einem vom Modell festgelegten Zeitpunkt während der Simulation werden für eine bestimmte Zeit Fehler aktiviert. Nach der erfolgten Fehlerinjektion wird die Simulation noch eine Weile fortgesetzt, um die Auswirkungen des Fehlers bestimmen zu können. Dieser Vorgang wird so lange mit unterschiedlichen Fehlertypen, Fehlerzeitpunkten und Fehlerdauern wiederholt, bis die vom Benutzer geforderte statistische Genauigkeit (Konfidenz) der zu berechnenden Systemgröße (z.B. Ausfallrate) erreicht ist.

#### 3.1.1 Simulation virtueller Systeme

Zur Simulation virtueller, digitaler Systeme werden in der Praxis weitgehend Ereignis-gesteuerte Simulatoren verwendet (z.B. **ADEPT** [Kumar95], **CSIM** [Schwetman86], **DEPEND** [Goswami90], **MEFISTO** [Jenn94], **SimPar** [Hein95]). Sie gestatten es, Abläufe in digitalen, deterministischen Systemen effizient zu simulieren.

Werden die Möglichkeiten dieser Sprachen korrekt eingesetzt, müssen zu bestimmten Zeitpunkten nur die Zustände und die Ausgangssignale von den Komponenten neu berechnet werden, deren Eingangssignale sich im vorhergehenden Simulationsschritt geändert haben. Zwischen verschiedenen Ereignissen brauchen keine Berechnungen durchgeführt zu werden. So ist es möglich, daß große Teile eines Systems, in denen sich während einer Zeitspanne keine Eingangssignale ändern und damit keine Ereignisse auftreten, während der Simulation keine Rechenzeit benötigen. Z.B. kann in einem simulierten RAM-Speicher jeweils nur auf ein einzelnes Speicherwort zugegriffen werden. Alle anderen Werte innerhalb des Speichers müssen nicht neu berechnet werden, da auf sie nicht zugegriffen wird.

Für einen VHDL-Simulator ist dies beispielsweise an den folgenden Sprachkonstruktionen ersichtlich (Abbildung 11, 12 und 13). Andere Ereignis-gesteuerte Simulatoren verwenden ähnliche Konstrukte zum Generieren bzw. Abfragen von Ereignissen.

```
ARCHITECTURE behaviour OF a
IS
    ...
    c <= ...;
    ...
END behaviour;
```

**Abb. 11: Signalisieren von Ereignissen in VHDL**

```
ARCHITECTURE behaviour OF a
IS
    ...
    WAIT ON c;
    b;
    ...
END behaviour;
```

**Abb. 12: Warten auf Ereignisse in VHDL**

```

ARCHITECTURE behaviour OF a
IS
    ...
    IF c'EVENT THEN
        b;
    ENDIF;
    ...
END behaviour;

```

**Abb. 13: Abfragen von Ereignissen in VHDL**

Durch eine Wertzuweisung an das Signal  $c$  wird im ersten Beispiel (Abbildung 11) ein Ereignis erzeugt. In den Beispielen der Abbildungen 12 und 13 sollen die Komponenten vom Typ  $a$  vor Ausführung von  $b$  warten, bis sich Signal  $c$  ändert, bzw. nur dann  $b$  durchführen, wenn sich das Signal geändert hat.

Wenn dynamische Mutanten<sup>1</sup> eingesetzt werden, kann jeweils dasselbe Simulationsprogramm für alle Fehlerinjektionen verwendet werden. In diesem Fall enthält das Programm alle fehlerfreien und alle fehlerhaften Komponenten (z.B. **REACT** [Clark93], **DEPEND** [Goswami90], **SimPar** [Hein95]). Im Falle statischer Mutanten muß immer, wenn ein anderer Fehler injiziert werden soll, eine neue, fehlerhafte Komponente und damit ein neues Simulationsprogramm erzeugt werden (z.B. **MEFISTO** [Jenn94]). Da die Zeit, um ein Simulationsprogramm zu generieren, weit über der eigentlichen Simulationszeit liegen kann, ist dies ein großer Nachteil der statischen Mutanten.

Zur genaueren, kontinuierlichen Simulation dieser Systeme auf Schaltungsebene werden hauptsächlich **SPICE**-ähnliche Simulatoren benutzt. Diese Simulatoren können auch analoge Signale nachbilden. Sie verwenden dazu i.a. Algorithmen der nicht-linearen Relaxation ([Saleh87], [Hennion85]). Ihre Handhabung ist jedoch im allgemeinen recht schwierig. Vom Benutzer sind zur Simulation z.B. (Zeit-) Schrittweiten anzugeben, von denen die Genauigkeit der Simulationsergebnisse bzw. die Simulationsdauer ganz erheblich abhängen kann. Dieser Parameter läßt sich durch Probieren verbessern. Im Falle von injizierten Fehlern kann sich ein System jedoch völlig anders verhalten als im Normalfall. Zum Beispiel können plötzlich Schwingungen auftreten, die eine völlig andere Einstellung der Schrittweite erfordern. Für eine große Zahl zu injizierender Fehler müssen daher viele Parametereinstellungen gefunden werden (manueller Aufwand) oder die Schrittweite muß sicherheitshalber sehr klein gewählt werden (hoher Simulationsaufwand). Derartige Simulationen eignen sich daher nur für kleine Modelle.

Die Ereignis-gesteuerte und die Zeit-kontinuierliche Simulation können zur Beschleunigung von Systemsimulationen gemeinsam verwendet werden (hybride Simulation, **FOCUS** [Choi92], [Yang92a]). In diesem Fall wird das digitale System in zwei Teile unterteilt. Der eine Teil (z.B. auf Register-Transfer- oder Gatterebene modelliert) wird wie gewöhnlich Zeit- und Werte-diskret simuliert. Er stimuliert und beobachtet den zweiten Teil, der mit analogen Signalwerten Zeit-kontinuierlich auf Schaltungsebene nachgebildet wird. Der Vorteil dieses Simulationsverfahrens liegt in der relativ großen Genauigkeit (Teilmodell auf Schaltungsebene) und der guten Simulationsgeschwindigkeit (Teilmodell auf Register-Transfer- oder Gatterebene). Leider muß das Modell jedoch zwischenzeitlich immer wieder verändert werden, um in verschiedenen Komponenten Fehler zu injizieren, da jeweils nur im Teilmodell auf Schaltungsebe-

1. durch Fehler modifizierte Komponenten

### 3. Effiziente Experimentdurchführung

---

ne Fehler genau simuliert werden können. Auch wenn sich diese Änderungen automatisieren lassen (ähnlich Abschnitt 3.7), verliert man durch sie wieder einen großen Teil der bei der eigentlichen Simulation gesparten Zeit.

#### 3.1.2 Simulation instrumentierter Systeme

Sollen instrumentierte Systeme bezüglich ihren Fehlertoleranzeigenschaften ausgewertet werden, „simulieren“ sie ihren normalen, fehlerfreien Ablauf selbst. Während der fehlerlosen Phase vor der Fehlerinjektion und in der Beobachtungsphase nach der Injektion läuft das System von der Instrumentierung unbeeinflusst. Nur während der eigentlichen Fehlerinjektion wird ein Teil des Systems mit Hilfe der Instrumentierung in einen im Normalfall nicht erreichbaren, fehlerhaften Zustand gezwungen.

##### Hardware-instrumentierte Systeme

Hardware-basierte Fehlerinjektoren (Beschreibung im Abschnitt 2.1) verwenden zusätzliche Hardware, um Fehler vorzutäuschen. Während der Fehlerinjektion muß der normale Systemablauf im Gegensatz zu den Software-implementierten Verfahren nicht unterbrochen werden. Das System kann daher durchgängig vor, während und nach der Fehlerinjektion mit der gleichen Geschwindigkeit wie im Original laufen. Dies ist ein großer Vorteil besonders bei Echtzeitsystemen, bei denen es nicht nur auf ein vom Wert sondern auch vom Zeitpunkt her richtiges Ergebnis ankommt. Daß die Fehlerinjektion keinen zusätzlichen Zeitaufwand erfordert, bedeutet weiterhin, daß auch die Injektion von permanenten Fehlern nicht mehr Zeit braucht als die Injektion von temporären Fehlern. Hier sind die Hardware-instrumentierten Systeme den Software-instrumentierten deutlich überlegen.

Bedingt durch parasitäre Kapazitäten durch die angefügten Kontakte einer Hardware-Instrumentierung (Pin-Level-Fehlerinjektion) kann es unter Umständen jedoch erforderlich sein, die Taktrate des untersuchten Systems herunterzusetzen. Dies ist schon bei heutigen Systemen ein großes Problem, das sich bei den zu erwartenden Leistungssteigerungen verbunden mit entsprechend erhöhten Taktraten der verfügbaren Rechner in den nächsten Jahren noch verschärfen dürfte.

Da die zusätzliche Hardware teuer ist, wird meist nur eine kleinere Anzahl von Pins instrumentiert (z.B. MESSALINE [Arlat90]: 32 Pins; RIFLE [Madeira94]: 96 Pins). So kann selbst in einem Multiprozessorsystem im allgemeinen nur ein einzelner Prozessor mit einer Instrumentierung versehen werden. Sollen Fehler über das gesamte System verteilt injiziert werden, ist es notwendig, die Steckkontakte entsprechend den zu simulierenden Fehlern manuell umzusetzen. Vollautomatische Auswertungen derartiger Modelle sind daher unmöglich bzw. sehr teuer. Ähnliches gilt für Fehlerinjektionen durch Bestrahlung mit Schwerionen. Die Ionenquelle muß entsprechend dem beabsichtigten Fehlerinjektionsort verschoben werden.

##### Software-instrumentierte Systeme

Alle Software-implementierten Fehlerinjektoren, die in Abschnitt 2.1 vorgestellt wurden, verwenden die Debug-Einrichtungen der in den entsprechenden Systemen eingesetzten Prozessoren. Dazu gehören der Einzelschrittmodus bzw. die Möglichkeit eine „Illegal Instruction Exception“ beim Erreichen einer bestimmten Programmzeile zu generieren, sowie die Fähigkeit der Systeme, nach bestimmten Zeiten durch Timer Unterbrechungen auszulösen. In der Ausnahmebehandlungsroutine können dann die Zustandsdaten des Systems ausgelesen und vor dem Rücksprung aus der Ausnahmebehandlungsroutine nahezu beliebig verfälscht werden. In nahezu allen UNIX-Systemen kann der Systemaufruf *ptrace*<sup>1</sup> für diesen Zweck herangezogen werden.

Die Software-Instrumentierung kann – bedingt durch ihren eigenen Ressourcenverbrauch – die Geschwindigkeit des Systems verringern. In der Praxis ist dies jedoch meist zu vernachlässigen, da die Fehlerinjektoren bei einer Größe von derzeit ca. 100 KByte bei heutigen Rechnern nicht mehr ins Gewicht fallen. Lediglich in kleinen Systemen (z.B. Embedded Controller) kann dies zu größeren Verzögerungen im Ablauf führen oder eine derartige Implementierung sogar gänzlich verhindern.

In einer Ausnahmebehandlungsroutine kann der Zustand des Systems nur einmalig manipuliert werden. Nach einem erfolgten Rücksprung aus dieser Routine läuft das System wieder völlig fehlerfrei weiter. Sollen länger andauernde oder sogar permanente Fehler simuliert werden, so muß die Ausnahmebehandlungsroutine für die Dauer des Fehlers nach jedem normalen Rechenschritt wiederholt ausgeführt werden. Dies kann zu einem Simulationsaufwand führen, der um mehrere Zehnerpotenzen über der normalen Systemlaufzeit liegt [Tschäche95]. Wird das laufende System dem Fehler entsprechend verändert [Lovric95] (z.B. alle Additionsbefehle durch entsprechende Unterprogrammaufrufe zur fehlerhaften Addition ersetzt), kann die Simulation von permanenten Fehlern deutlich beschleunigt werden. Es erscheint jedoch sehr fraglich, ob dieses Verfahren für eine größere Anzahl verschiedener Fehler praktikabel ist, da der Aufwand auch nur einen einzelnen Fehlertyp zu implementieren, sehr groß und zudem nicht portabel ist. Vor und nach der eigentlichen Fehlerinjektion ist der Ablauf der Ereignisse im allgemeinen vom Fehlerinjektor unbeeinflusst und damit – im Gegensatz zu einer Simulation – immer genauso schnell wie im Original.

Bei der Simulation von Fehlern ergibt sich als weiterer Nachteil der Software-instrumentierten Systeme, daß eingestreute Fehler u.U. nicht nur das zu testende System, sondern auch die Beobachtungs-Software stören können und somit die Vertrauenswürdigkeit der Ergebnisse nicht garantiert werden kann. Dieser Nachteil dieses Verfahrens wird in keiner der bekannten Veröffentlichungen erwähnt. Die Wahrscheinlichkeit, daß z.B. in einem UNIX-System ein zu untersuchender Prozeß den überwachenden Prozeß stört, ist bedingt durch die Speicherschutzmechanismen zwar klein, jedoch nicht null. Zum Beispiel kann ein Prozeß durch Signale und Ein- und Ausgabeoperationen seine Umgebung und damit andere Prozesse beeinflussen. In Systemen ohne Speicherschutzmechanismen ist eine Beeinflussung sogar relativ wahrscheinlich, da viele der injizierten Fehler sich durch fehlerhafte Speicherzugriffe bemerkbar machen (siehe z.B. [Sieh94]).

## 3.2 Einzelfehlerannahme

Im folgenden wird davon ausgegangen, daß die modellierten Fehler voneinander statistisch unabhängig sind. Ist dies nicht der Fall („Common Cause Fault“), muß das Modell so umgestellt werden, daß die modellierten Fehler keine gemeinsame Ursache mehr haben. Z.B. kann ein Fehler in der Stromversorgung zum Ausfall mehrerer oder sogar aller Gatter eines Computersystems führen. Soll dieser Fehler modelliert werden, sollte jedes Gatter einen Versorgungseingang haben, der mit dem (fehlerhaften) Netzteil verbunden ist und entsprechend auf Änderungen in der Stromversorgung mit fehlerhaften Ausgängen/Zuständen reagieren. Ähnlich können andere zu Fehlern führende Einflüsse der Umgebung (z.B. Wärme, Strahlung usw.) extra modelliert und als Parameter für die Funktion der Komponenten herangezogen werden. Ein anderes Beispiel für eine Fehlerursache, deren Auswirkungen sich an einer Vielzahl von Gatterausgängen zeigen können, ist ein Kurzschluß zwischen zwei verschiedenen Signalleitungen („Cur-

---

1. siehe UNIX-Manual „ptrace(2)“

### 3. Effiziente Experimentdurchführung

---

rent-“ oder „Logic-Large-Scope-Short“, siehe [Jacomet91]). In diesem Fall ist der Kurzschluß als eigentliche Fehlerursache zu modellieren, während die fehlerhaften Werte an den nachfolgenden Gatterausgängen dessen Auswirkungen sind.

Sind die modellierten Fehler voneinander statistisch unabhängig und ist die Wahrscheinlichkeit klein, daß überhaupt ein Fehler während der Simulationszeit vorhanden ist, kann der Fall, daß während einer Simulation mehrere Fehler auftreten, im allgemeinen vernachlässigt werden. Das Ergebnis eines Simulationslaufes ohne Fehler ist durch den einmal zu berechnenden Golden-Run bekannt. Es bleiben daher nur noch die Fälle experimentell zu untersuchen, bei denen während der Simulation genau ein Fehler auftritt.

In nahezu allen Veröffentlichungen über Fehlerinjektionsexperimente wird davon ausgegangen, daß nur einzelne Fehler während des Betriebs eines Systems auftreten. Die Autoren nehmen jedoch alle eine Gleichverteilung der einzelnen Fehler (z.B. interne Stuck-at- und Pin-Level-Fehler) an, was in der Praxis normalerweise nicht der Fall ist. So ist zum Beispiel bekannt, daß Fehler in der Nähe der nach außen führenden Pins gegenüber Fehlern in weiter innen liegenden Bereichen überproportional häufig auftreten. Im folgenden soll gezeigt werden, wie Fehler zu injizieren sind, wenn nicht von einer Gleichverteilung ausgegangen werden kann.

Dabei sind zwei verschiedene Ansätze denkbar. Die erste Möglichkeit besteht darin, Fehler so zu injizieren, daß die relative Häufigkeit der Fehler proportional ist zu den entsprechenden Auftretswahrscheinlichkeiten der Fehler. Dieser Ansatz wird im nächsten Abschnitt (3.2.1) besprochen. Der darauffolgende Abschnitt 3.2.2 zeigt die zweite Möglichkeit. Dort werden Überlegungen angestellt, Fehler so zu injizieren, daß das gewünschte Analyseergebnis (z.B. die Ausfallrate des Systems) bei gegebener Anzahl von zu injizierenden Fehlern möglichst genau wird. Wie sich zeigt, führen diese beiden Ansätze zum gleichen Ergebnis. Abschnitt 3.2.3 beschreibt, wie die Fehlerinjektionsexperimente gemäß den theoretischen Überlegungen in den Abschnitten 3.2.1 und 3.2.2 in der Praxis durchgeführt werden können.

#### 3.2.1 Fehlerinjektion gemäß der relativen Häufigkeit der Fehler

Sind für die verschiedenen, möglichen Fehler  $f \in F$  die Raten  $\lambda_f$  (Exponentialverteilung) bekannt, so ergibt sich für jeden Simulationslauf der simulierten Zeitdauer  $T$  für die einzelnen Fehler die Auftretswahrscheinlichkeit  $P_f$  nach folgender Formel:

$$P_f = 1 - e^{-\lambda_f T}.$$

Für die Wahrscheinlichkeit, daß genau ein Fehler während einer simulierten Zeit  $T$  auftritt, ergibt sich unter der Annahme, daß alle Fehler voneinander unabhängig sind:

$$P(\text{genau ein Fehler}) = e^{-\sum_{f \in F} \lambda_f T} \sum_{f \in F} \frac{1 - e^{-\lambda_f T}}{e^{-\lambda_f T}}$$

Ist

$$\sum_{f \in F} \lambda_f T \ll 1$$

und damit auch  $\lambda_f T \ll 1$  für alle  $f \in F$ , erhält man für  $P(\text{genau ein Fehler})$  folgende Näherung:

$$P(\text{genau ein Fehler}) = \sum_{f \in F} \lambda_f T.$$

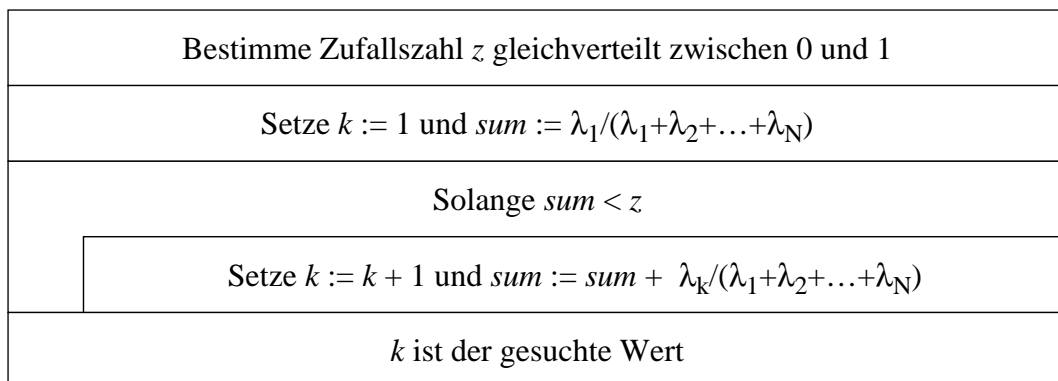
Für die bedingte Wahrscheinlichkeit, daß Fehler  $k$  aufgetreten ist, wenn bekannt ist, daß genau ein Fehler aktiv ist, ergibt sich daraus:

$$P(\text{Fehler } k \mid \text{genau ein Fehler}) = \frac{\lambda_k}{\sum_{f \in F} \lambda_f}.$$

Für den bedingten Erwartungswert der Anzahl  $n_k$  der Fehler vom Typ  $k$ , die bei  $N$  durchgeführten Fehlerinjektionsexperimenten auftreten, erhält man damit:

$$n_k = \frac{\lambda_k}{\sum_{f \in F} \lambda_f} N.$$

Soll für die Simulation ein Fehler  $k$  per Zufall entsprechend den gegebenen Raten ausgewürfelt werden, kann dies durch folgenden Algorithmus erreicht werden (siehe Abbildung 14).



**Abb. 14: Algorithmus zur zufälligen Auswahl eines Fehlers**

#### 3.2.2 Fehlerinjektion mit hoher Konfidenz der Ergebnisse

Gesucht ist die Antwort auf die Frage, welcher Fehler wie oft injiziert werden muß, damit das gesuchte Ergebnis der Auswertung eine möglichst große Konfidenz erreicht. Besitzen alle Fehler des Modells die gleiche Rate, müssen alle Fehler gleich häufig injiziert werden, da die Fehler untereinander keinerlei Priorität besitzen. Treten jedoch bestimmte Fehler häufiger auf als andere, ist anzunehmen, daß ihre Auswirkungen mit größerer statistischer Genauigkeit bestimmt werden müssen als die derjenigen Fehler, die nur selten vorkommen.

Dies soll im folgenden am Beispiel der Ausfallrate eines Systems gezeigt werden. Berechnungen für die Recovery-Zeit-Verteilung, Ausfallwahrscheinlichkeit in einem bestimmten Zeitintervall usw. verlaufen ähnlich.

Die Ausfallrate eines Systems kann durch die folgende Formel angegeben werden:

$$\mu = \sum_{f \in F} \lambda_f p_f$$

( $F$ : Menge der möglichen, verschiedenen Fehler im System,  $\lambda_f$ : Rate des Fehlers  $f$ ,  $p_f$ : Wahrscheinlichkeit, daß das System aufgrund des Fehlers  $f$  ausfällt).

### 3. Effiziente Experimentdurchführung

Nach dem Gauß'schen Fehlerfortpflanzungsgesetz ([Gotthardt68]) gilt für den mittleren Fehler eines Funktionswertes  $\Delta(y)$  mit  $y = f(x_0, x_1, \dots, x_{N-1})$ :

$$\Delta(y)^2 = \left( \frac{\partial f}{\partial x_0} \right)^2 \Big|_{x_0=0} \Delta(x_0)^2 + \left( \frac{\partial f}{\partial x_1} \right)^2 \Big|_{x_1=0} \Delta(x_1)^2 + \dots + \left( \frac{\partial f}{\partial x_{N-1}} \right)^2 \Big|_{x_{N-1}=0} \Delta(x_{N-1})^2,$$

wenn die Parameter  $x_i$  einen mittleren Fehler von  $\Delta(x_i)$  besitzen.

Für die Ausfallrate gilt daher:

$$\Delta(\mu)^2 = \sum_{f \in F} \lambda_f^2 \Delta(p_f)^2.$$

Außerdem gilt die Tschebyscheffsche Ungleichung ([Bronstein87]) für jedes  $\varepsilon > 0$ :

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

mit  $X$ : Zufallsvariable,  $EX$ : Erwartungswert der Zufallsvariablen und  $DX$ : Varianz der Zufallsvariablen.

Für die Binomialverteilung mit den Parametern  $p$  und  $n$  gilt  $EX = np$  und  $DX = np(1-p)$ . Somit erhält man mit der relativen Häufigkeit  $m/n$  eines Ereignisses:

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

Dies bedeutet, daß die Wahrscheinlichkeit, daß eine gemessene relative Häufigkeit  $m/n$  eines Ereignisses von der realen Auftretswahrscheinlichkeit  $p$  um mehr als den Betrag  $\varepsilon$  abweicht, kleiner ist als  $1/(4\varepsilon^2 n)$  (bei  $n$  Tests).

Für einen mittleren Fehler  $\Delta(X)$  gilt allgemein:

$$P(|X - EX| \geq \Delta(X)) = K \text{ mit } K = 1 - \int_{-\sigma}^{\sigma} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx \approx 0.3174.$$

Als oberer Schätzwert für den mittleren Fehler  $\Delta(p_f)$  kann hier daher gelten:

$$\frac{1}{4\Delta(p_f)^2 n_f} = K \text{ bzw. } \Delta(p_f)^2 = \frac{1}{4Kn_f}.$$

Damit folgt für den mittleren Fehler  $\Delta(\mu)$  der Ausfallrate  $\mu$

$$\Delta(\mu)^2 \leq \sum_{f \in F} \lambda_f^2 \frac{1}{4Kn_f} = \frac{1}{4K} \sum_{f \in F} \lambda_f^2 \frac{1}{n_f}.$$

Das heißt, daß Fehlerinjektionsexperimente mit  $N$  einzelnen Fehlerinjektionen dann besonders genaue Ergebnisse bezüglich der Ausfallrate eines Systems ergeben, wenn

$$\Delta(\mu)^2 = \frac{1}{4K} \sum_{f \in F} \lambda_f^2 \frac{1}{n_f} \text{ minimal ist und die Nebenbedingung } \sum_{f \in F} n_f = N \text{ gilt.}$$

Hier ist also das Minimum der Funktion  $\Delta(\mu)^2$  für ganzzahlige Werte  $n_f$  mit der Nebenbedingung

$$\sum_{f \in F} n_f = N$$

zu finden. Leider existiert i.a. keine geschlossene Lösungsformel für dieses Problem. Daher wird im folgenden das Minimum der Funktion  $\Delta(\mu)^2$  für reelle Zahlen gesucht. Werden die so gefundenen reellen Zahlen  $n_f$  jeweils auf die nächstgrößere ganze Zahl aufgerundet, wird der mittlere Fehler für  $\mu$  nicht schlechter. Es werden jedoch mehr Experimente durchgeführt, als notwendig gewesen wären (siehe dazu auch Abschnitt 3.2.3).

Setzt man die Nebenbedingung in die Formel für  $\Delta(\mu)^2$  ein, ergibt sich:

$$\Delta(\mu)^2 = \frac{1}{4K} \left( \sum_{f \in F - \{l\}} \lambda_f^2 \frac{1}{n_f} + \lambda_l^2 \frac{1}{N - \sum_{f \in F - \{l\}} n_f} \right)$$

Notwendige Voraussetzung für ein Minimum dieser Funktion für einen Parametervektor

$$(n_{f_1}, n_{f_2}, \dots, n_{f_{|F|}})$$

ist, daß alle partiellen Ableitungen von  $\Delta(\mu)^2$  für diesen Vektor den Wert 0 besitzen. Die partiellen Ableitungen berechnen sich wie folgt:

$$\frac{\partial}{\partial n_k} \Delta(\mu)^2 = \frac{1}{4K} \left( -\frac{\lambda_k^2}{n_k^2} + \frac{\lambda_l^2}{\left( N - \sum_{f \in F - \{l\}} n_f \right)^2} \right) \text{ mit } k \in F - \{l\} .$$

Mit dem Ansatz

$$n_k = \frac{\lambda_k}{\sum_{f \in F} \lambda_f} N$$

ergibt sich

$$\frac{\partial}{\partial n_k} \Delta(\mu)^2 = \frac{1}{4K} \left( -\frac{\left( \sum_{f \in F} \lambda_f \right)^2}{N^2} + \frac{\lambda_l^2}{\left( N - \frac{N}{\sum_{f \in F} \lambda_f} \sum_{f \in F - \{l\}} \lambda_f \right)^2} \right)$$

und weiter

$$\frac{\partial}{\partial n_k} \Delta(\mu)^2 = \frac{1}{4K} \left( -\frac{\left( \sum_{f \in F} \lambda_f \right)^2}{N^2} + \frac{\left( \sum_{f \in F} \lambda_f \right)^2}{N^2} \right) = 0 .$$

### 3. Effiziente Experimentdurchführung

---

Durch obige Rechnung ist gezeigt, daß

$$n_k = \frac{\lambda_k}{\sum_{f \in F} \lambda_f} N$$

die notwendige Bedingung für ein Minimum der Funktion  $\Delta(\mu)^2$  erfüllt. Da alle Terme  $\lambda_f^2/n_f$  mit  $n_f$  mit  $0 \leq n_f \leq \infty$  fallen, das globale Fallen der Funktion  $\mu$  aber durch

$$\sum_{f \in F} n_f = N$$

beschränkt ist, kann es nur ein einziges Minimum der Funktion  $\Delta(\mu)^2$  mit  $0 \leq n_f$  und

$$\sum_{f \in F} n_f = N$$

geben (hinreichende Bedingung). Berechnungen zur Bestimmung der Recovery-Zeit-Verteilung, der Wahrscheinlichkeit eines Ausfalls in einem gegebenen Zeitintervall u.ä. ergeben die gleichen Werte für die einzelnen Werte  $n_f$ .

Daraus folgt:

Sollen  $N$  Fehler injiziert werden, so sollte jeder Fehler  $k$  mit der Häufigkeit

$$n_k = \frac{\lambda_k}{\sum_{f \in F} \lambda_f} N$$

injiziert werden. Dann ist das Ergebnis (im Beispiel die Ausfallrate  $\mu$ ) am genauesten bestimmt. Dieser Wert für  $n_k$  stimmt mit dem Ergebnis aus Abschnitt 3.2.1 überein.

#### 3.2.3 Praktische Durchführung der Experimente

Die in den obigen Abschnitten gewonnene, theoretische Erkenntnis, daß die Anzahl der Fehlerinjektionen  $n_k$  jedes Fehlers  $k$

$$n_k = \frac{\lambda_k}{\sum_{f \in F} \lambda_f} N$$

betragen sollte, läßt sich nicht ohne weiteres in die Praxis umsetzen, da die so gewonnenen Werte  $n_k$  im allgemeinen keine ganzen Zahlen sind. Die meisten werden einen Wert knapp über 0 besitzen, da in der Praxis die Anzahl der Fehlerinjektionen sehr viel kleiner ist als die Anzahl der möglichen verschiedenen Fehler ( $N \ll |F|$ ). Es verbietet sich, die Werte für die einzelnen  $n_k$  auf die nächstgrößere ganze Zahl aufzurunden, da dann die Anzahl der zu injizierenden Fehler die Größe der Menge der verschiedenen Fehler erreichen kann und diese Zahl im allgemeinen um mehrere Größenordnungen zu groß ist, um so viele Fehler zu injizieren.

Bei der Modellierung von größeren Systemen kommen jedoch meist nur eine kleine Menge verschiedener Fehlerraten vor, da die Systeme aus einer normalerweise recht kleinen Menge von verschiedenen Typen von Standardkomponenten<sup>1</sup> zusammengesetzt werden (siehe Abschnitt 2.5) und alle gleichartigen Komponenten gleiche Fehlerraten besitzen. Werden derartige Men-

gen von Fehlern mit gleicher Fehlerrate zu einer Fehlerklasse zusammengefaßt, so läßt sich ähnlich den obigen Ableitungen für die einzelnen Fehler die Anzahl der Injektionen innerhalb der einzelnen Fehlerklassen bestimmen. Die Fehler innerhalb der einzelnen Fehlerklassen sind dann gleichverteilt, da sie alle die gleiche Rate besitzen.

Sei

$$\{f_1, f_2, \dots, f_m\} = \bigcup_{k \in K} G_k$$

eine Klasseneinteilung der verschiedenen Fehler  $f_1$  bis  $f_m$  gemäß der obigen Erläuterung. Dann gilt:

$$\forall i \neq j: G_i \cap G_j = \emptyset \text{ (disjunkte Fehlerklassen)}$$

und

$$f_i \in G_k \wedge f_j \in G_k \Rightarrow \lambda_{f_i} = \lambda_{f_j} \text{ (Fehler einer Klasse besitzen die gleiche Rate).}$$

Dann ergibt sich für die Fehlerrate  $\lambda_{G_i}$  der Fehlerklasse  $G_i$ :

$$\lambda_{G_i} = \sum_{f \in G_i} \lambda_f.$$

Entsprechend obigen Überlegungen für die Anzahl der zu injizierenden Einzelfehler ergibt sich für die Anzahl der Injektionen  $n_{G_i}$  innerhalb der Klasse  $G_i$ :

$$n_{G_i} = \frac{\lambda_{G_i}}{\sum_{f \in F} \lambda_f} N \text{ oder } n_{G_i} = \frac{\sum_{f \in G_i} \lambda_f}{\sum_{f \in F} \lambda_f} N.$$

Sollten sich für die Werte  $n_{G_i}$  wiederum nicht-ganzzahlige Werte ergeben, können diese jeweils auf die nächstgrößere, ganze Zahl aufgerundet werden. Da es nur eine relativ kleine Anzahl von Klassen gibt, wird sich die Anzahl der zu injizierenden Fehler damit nur geringfügig erhöhen (im Durchschnitt etwa um die Hälfte der Anzahl der Klassen). Die Konfidenz des Ergebnisses wird sich nicht verschlechtern. Die Aufteilung der  $n_{G_i}$  zu injizierenden Fehler der Klasse  $G_i$  erfolgt dann jeweils gleichverteilt, da alle Fehler einer Klasse die gleiche Rate besitzen.

## 3.3 Simulation mit Hilfe des Zielsystems

### 3.3.1 Prinzipielles Verfahren

Häufig sind Fehler in untersuchten Systemen nur sehr kurze Störungen, die wenige Mikrosekunden oder noch kürzer andauern. Im Falle von direkten Bit-Flips sind die Fehler sogar gänzlich zeitlos. Die eigentliche Fehlersimulation ist bei der Untersuchung von Fehlertoleranzeigenschaften daher meist eine schnell ausgeführte Aufgabe. Viel aufwendiger ist dagegen die Phase vor der eigentlichen Fehlersimulation, die notwendig ist, um den Ausgangspunkt der Fehlersi-

---

1. Die umfangreichste, mit dem bekannten, kommerziellen Synthese-Werkzeug von SYNOPSIS mitgelieferte Bibliothek enthält derzeit 184 verschiedene Standardkomponenten.

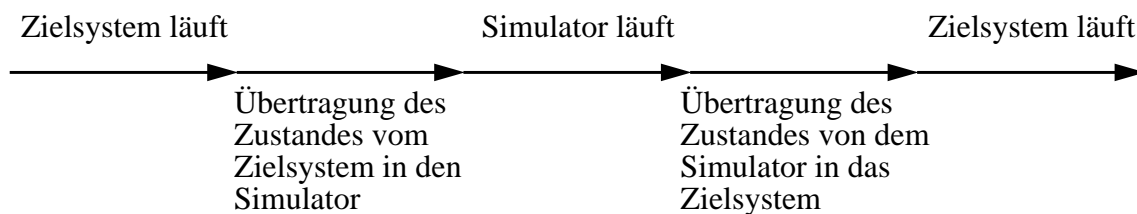
### 3. Effiziente Experimentdurchführung

mulation zu berechnen. Auch die Zeit, die zur Bestimmung der Fehlerauswirkungen benötigt wird, übersteigt in den meisten Fällen die Fehlersimulationszeit um mehrere Größenordnungen. Daraus folgt, daß im Falle temporärer Fehler ein besonderes Augenmerk der Simulationsgeschwindigkeit vor und nach der Fehlersimulationsphase gelten sollte. Eine Ausnahme bilden die permanenten Fehler. In diesem Fall kann durch das im folgenden beschriebene Verfahren nur die Simulation vor der Fehlerinjektion beschleunigt werden.

Zur Simulation der fehlerlosen Phasen bietet sich – sofern sie existiert – die Hardware des zu untersuchenden Systems selbst an. Sie kann das Modell optimal „simulieren“. Da keinerlei Fehler einzustreuen sind, müssen auch keine Fehler-injizierenden Komponenten eingebaut sein. Je nach gewünschter Auswertung kann jedoch eventuell eine Hard- oder Software-Instrumentierung zur Messung von Fehlerlatenzzeiten u.ä. notwendig werden.

Ein einzelnes Fehlerinjektionsexperiment sieht dann wie folgt aus:

Zunächst wird auf der Hardware das System im Original gestartet. Nach einiger Zeit, zum Fehlerinjektionszeitpunkt, wird der gesamte Zustand des Systems (Speicher- und Registerinhalte) von der Hardware zum Simulator übertragen. Der Simulator berechnet dann aus diesem Zustand einen durch den aktivierten Fehler veränderten Folgezustand. Im Falle temporärer Fehler wird dieser auf das Originalsystem zurückkopiert und die Hardware ermittelt daraus dann die Auswirkungen des Fehlers. Sollen permanente Fehler simuliert werden, verbleibt der Zustand im Simulator und dieser protokolliert die Auswirkungen des Fehlers.



**Abb. 15: Simulation mit Hilfe des Zielsystems**

Schwierigkeiten bestehen bei diesem Verfahren darin, den Zustand der Hardware vor der Fehlersimulation auszulesen und im Falle temporärer Fehler danach gemäß dem injizierten Fehler zu manipulieren. Dies ist jedoch vergleichbar mit den Fähigkeiten von Software-implementierten Fehlerinjektoren (z.B. FERRARI [Kanawati92], FIAT [Segall88][Barton90], ProFI [Lovric95], XCEPTION [Carreira95b]). Eine weitere Lösung bieten sogenannte In-Circuit-Emulatoren, die statt der normalen CPU des Systems in deren Sockel gesteckt werden können. Diese Emulatoren verhalten sich wie die normale CPU, die sie ersetzen, können aber zusätzlich beliebig gestartet und gestoppt werden. Außerdem erlauben sie es, den Zustand der CPU zu beliebigen Zeiten zu lesen und zu manipulieren. Über die CPU kann darüber hinaus durch normale Schreib- und Lesezugriffe der Speicher sowie die Register der Ein- und Ausgabegeräte erreicht werden. Das heißt, daß auf alle dem Programmierer des Systems zugänglichen Zustandsdaten des Systems zugegriffen werden kann. In anderen Registern (z.B. Pipeline-Register, Cache usw.) liegende Daten können nicht verwendet werden oder sind nur mühsam zu lesen bzw. zu setzen.

Um den Übertragungsaufwand (der Zustand eines Systems kann in heutigen Systemen mehrere Gigabytes umfassen) in Grenzen zu halten, ist die folgende Lösung denkbar: Während der relativ kurzen Fehlersimulation werden nur sehr wenige der Zustandsdaten benötigt. In der kurzen Zeit können eine CPU oder ein Ein-/Ausgabegerät nur auf wenige Register und Speicherzellen zugreifen. Es ist daher nicht notwendig, den gesamten Register- und Speicherinhalt des Systems

in den Simulator zu übertragen. Sinnvoll ist es, sich statt dessen zu merken, welche Speicherzellen bisher kopiert wurden. Greift der Simulator auf ein bisher nicht übertragenes Datum zu, wird die Simulation solange angehalten, bis der Wert aus der Hardware ausgelesen wurde. Ähnlich brauchen nur die Zustandsdaten auf die Hardware zurückübertragen werden, die verändert wurden. Dieses Verfahren ist vergleichbar mit dem „Demand Paging“ [Deitel90]. Auch dort werden nur dann Seiten eines Programms vom Hintergrundspeicher in den Hauptspeicher übertragen, wenn sie wirklich gebraucht werden, sofern auf sie zugegriffen wird („Read on Demand“).

Der Vorteil dieses Verfahrens gegenüber den Software-implementierten Verfahren liegt darin, daß die Modellierung der Fehler deutlich einfacher bzw. sehr viel detaillierter möglich ist. In der Literatur werden nur Fehler vorgestellt, die einfach zu modellieren sind (z.B. Bit-Flip, Nachrichtenverlust usw.). Fehler z.B. der ALU fehlen entweder vollständig oder sind nur exemplarisch implementiert. Dies liegt daran, daß für diese Zweck ungeeignete Programmiersprachen (meist C), nicht aber Hardware-Beschreibungssprachen (wie z.B. VHDL) verwendet werden. Durch das hier beschriebene Verfahren können alle in Kapitel 2 beschriebenen Probleme der Software-implementierten Fehlerinjektionsverfahren bezüglich der Modellierung vermieden werden.

#### 3.3.2 Bewertung des Verfahrens

Die Zeit  $t$ , die für ein einzelnes Fehlerinjektionsexperiment aufgewendet werden muß, läßt sich wie folgt berechnen:  $t = t_v + t_a + t_i + t_s + t_b$  mit  $t_v$ : Zeit, die benötigt wird, den Zustand vor der Fehlerinjektion zu berechnen;  $t_a$ : Zeit für das Auslesen des Systemzustandes;  $t_i$ : Zeit für die eigentliche Fehlerinjektion;  $t_s$ : Zeit für das Zurückschreiben des Zustandes;  $t_b$ : Beobachtungszeit. Es gilt  $t \approx t_v + t_b$ , da die Zeit für die Übertragung des Zustandes von der Hardware zum Simulator bzw. umgekehrt geringfügig ist im Vergleich zur Zeit, die für die Berechnung des Fehlersimulationsausgangszustandes bzw. zur Beobachtung aufgewendet werden muß. Auch die Fehlersimulationszeit ist i.a. sehr viel kürzer als die Zeit, die für die Berechnung des Ausgangszustandes für die Fehlersimulation benötigt wird.

Wird ein Problem, das  $T$  Rechenschritte erfordert, auf einer Hardware der Rechengeschwindigkeit „ $N$  Rechenschritte pro Zeiteinheit“ gelöst, benötigt diese  $t = T/N$  Zeiteinheiten dafür. Rechnet hier die Hardware  $A$  (Rechengeschwindigkeit  $N_A$ ) selbst während der Zeit vor der Fehlerinjektion, benötigt sie dafür  $t_v = T_v/N_A$  Zeiteinheiten. Wird die Hardware durch ein System  $B$  (Rechengeschwindigkeit  $N_B$ ) simuliert, gilt:  $t_v' = (T_v K)/N_B$ .  $K$  ist dabei ein Faktor, der angibt, wieviele Rechenschritte im Simulator auszuführen sind, um einen einzelnen Rechenschritt der zu simulierenden Hardware nachzubilden. Entsprechendes gilt für  $t_b$ . Die Beschleunigung der Experimente ist daher  $F = t_v'/t_v = t_b'/t_b = (KN_A)/N_B$ .

Die Konstante  $K$  wurde in [Tschäche95] und [Riecken95] je an einem Beispiel gemessen. Dort gilt jeweils  $K \approx 100$  für eine Simulation auf Verhaltensebene. Auch der in [Sieh94] beschriebene Simulator zeigte bei Messungen Werte für  $K$  in dieser Größenordnung. Simulationen auf Gatterebene oder gar Schaltungsebene können jedoch weit größere Werte von  $K$  ergeben. Die Konstante ist weiterhin selbstverständlich von der Qualität des verwendeten Simulators abhängig.

Insgesamt ergibt sich, daß das hier beschriebene Verfahren um so attraktiver ist, je leistungsfähiger das zu simulierende System, je langsamer der benutzte Simulator bzw. je genauer das Modell ist. Wenn die verwendeten Systeme in etwa gleich schnell sind, kann man mit Beschleunigungen um mehrere Größenordnungen rechnen.

### 3. Effiziente Experimentdurchführung

Leider ist das Verfahren jedoch nur für Systeme geeignet, von denen zumindest ein Prototyp existiert, da die Hardware selbst zur Simulation verwendet wird. Unter Umständen kann auch eine Hardware eingesetzt werden, deren Programmiermodell mit dem des zu untersuchenden Systems übereinstimmt. Das heißt, daß die Systeme zueinander Software-kompatibel sein müssen. In diesem Fall ist es jedoch meist aufwendiger, die beiden Zustände ineinander umzurechnen (z.B. wenn unterschiedlich große Prozessor-Caches oder verschiedene Befehls-Pipelines verwendet werden).

Schwierigkeiten können sich auch bei Realzeitanwendungen ergeben. Bei sehr harten Zeitbedingungen können durch die zwischenzeitliche Simulation und durch die Datenübertragungszeiten – auch ohne eingestreuten Fehler – Zeitbedingungen verletzt werden.

Die in [Sieh94] vorgestellte Arbeit zeigt, daß das vorgestellte Verfahren praktikabel ist und zu sehr detaillierten Ergebnissen kommt.

## 3.4 Multi-Threaded Fault-Injection

### 3.4.1 Prinzipielles Verfahren

Soll zu einem Zeitpunkt  $t_i$  ein Fehler in ein System injiziert werden, muß zunächst der Zustand  $Z(t_i)$  des Systems zu diesem Zeitpunkt festgestellt werden. Erst dann kann die eigentliche Fehlerinjektion erfolgen. Der berechnete Zwischenzustand wird als Ausgangspunkt der Fehlersimulation und der nachfolgenden Beobachtungsphase verwendet. Um nachfolgend weitere Fehler zu späteren Zeiten  $t_j$  ( $t_i < t_j$ ) simulieren zu können, müssen neue, fehlerfreie Zwischenzustände  $Z(t_j)$  berechnet werden (Einzelfehlerannahme, siehe Abschnitt 3.2). Dazu ist es jedoch nicht notwendig, das System von Startzustand aus neu zu simulieren. Es kann statt dessen vom vorher berechneten Zwischenzustand vom Zeitpunkt  $t_i$  ausgegangen werden (siehe Abbildung 16).

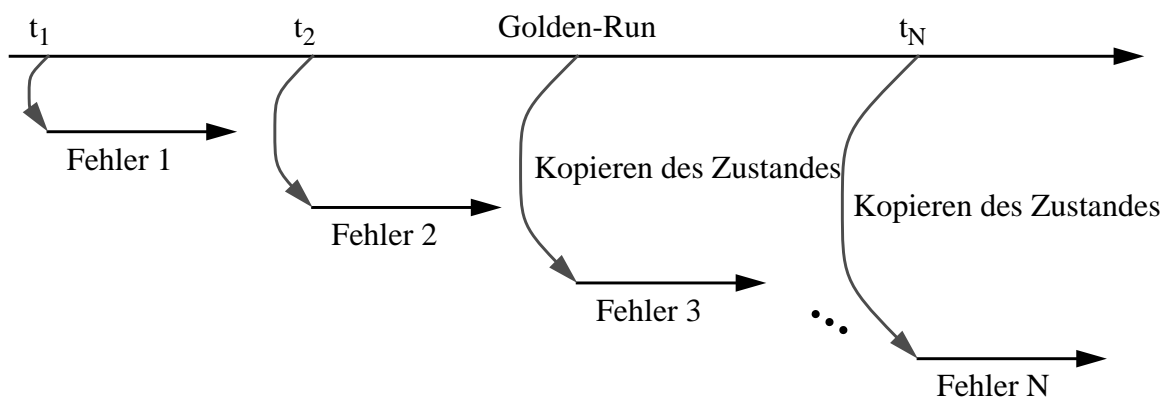


Abb. 16: Multi-Threaded Fehlerinjektion

Die Berechnung der Zwischenzustände erfolgt ohne Fehlerinjektionen. Dieser Simulationslauf kann daher gleichzeitig als Golden-Run zum Vergleich der Ergebnisse mit und ohne Fehlerinjektionen dienen.

Während des Golden-Runs müssen immer wieder einzelne Zwischenzustände für Fehlerinjektionsläufe kopiert werden. Es ist jedoch nicht notwendig, den gesamten Zustand des Systems tatsächlich zu kopieren. Es ist statt dessen sinnvoll, eine logische Kopie des Zustandes anzulegen. Erst wenn sich der Zustand durch Schreibzugriffe während der weiteren Simulation ändert,

sollten die entsprechenden Teilzustände wirklich kopiert werden. Dieses Verfahren ist unter dem Namen „Copy-on-Write“ bekannt [Deitel90]. Da während der Fehlersimulation nur auf einen relativ kleinen Teil des Gesamtzustandes zugegriffen wird (vergleiche Abschnitt 3.3), kann dadurch viel Kopieraufwand gespart werden.

### 3.4.2 Bewertung des Verfahrens

Sollen zu  $N$  Zeitpunkten  $t_1$  bis  $t_N$  (mit  $t_1 \leq t_2 \leq \dots \leq t_N$ ) Fehler injiziert werden, müssen - ohne Einsatz dieses Verfahrens -  $t_1 + t_2 + \dots + t_N$  Zeiteinheiten simuliert werden, um die jeweiligen Ausgangspunkte für die Fehlerinjektionen zu erhalten. Wird oben beschriebener Algorithmus verwendet, um die Zwischenzustände zu generieren, muß nur noch einmal bis zum Zeitpunkt  $t_N$  simuliert werden. Geht man davon aus, daß das Kopieren der Zwischenzustände sehr viel weniger Zeit benötigt als die Simulation, ergibt sich damit ein Faktor  $F$  der Zeitersparnis bei der Generierung der Zwischenzustände zu

$$F \approx \frac{t_1 + t_2 + \dots + t_N}{t_N}.$$

Liegen die Zeiten  $t_1$  bis  $t_N$  gleichverteilt im Intervall  $[0, t_N]$ , so ergibt sich für  $F$  im Mittel:

$$F \approx \frac{N \frac{1}{2} t_N}{t_N} = \frac{N}{2}.$$

Da die Zeit, die zur Berechnung der Zwischenzustände aufzuwenden ist, meist sehr viel größer ist als die eigentliche Fehlersimulations- und -beobachtungszeit, ist dies gleichzeitig der Faktor, mit dem Fehlerinjektionsexperimente beschleunigt werden können. Da im allgemeinen 1000 und mehr Fehler injiziert werden müssen, um eine ausreichende Konfidenz der Ergebnisse zu erhalten, ist dies eine sehr beachtliche Möglichkeit zur Leistungssteigerung von Fehlerinjektionsexperimenten.

## 3.5 Parallele Simulation

### 3.5.1 Prinzipielles Verfahren

Die einzelnen Simulationsläufe können parallel auf verschiedenen Rechenknoten durchgeführt werden, da sie voneinander unabhängig sind. Die zu erwartende Beschleunigung wird bei  $N$  Simulationsknoten auch bei einem Wert nahe  $N$  liegen. Der Mehraufwand liegt nur in der Verteilung der Simulationsparameter (Beschreibungen der einzelnen Fehlerinjektionen), die einmal am Anfang der Simulation durchzuführen ist, sowie in der Zusammenfassung der Ergebnisse (Fehlerlatenz-, Recovery-Zeiten, u.ä.) nach Ablauf aller Simulationen. Es ist zu erwarten, daß sowohl das Verteilen der Parameter als auch das Einsammeln der Ergebnisse im Vergleich zur eigentlichen Simulation sehr viel schneller erfolgt, da die zu verschickende Datenmenge relativ klein ist.

In Verbindung mit dem Verfahren der Multi-Threaded-Fehlerinjektion (Abschnitt 3.4) sind folgende zwei Verfahrensweisen denkbar:

1. Ein ausgezeichneter Knoten berechnet den Golden-Run und damit die Zwischenzustände, von denen aus die Fehlerinjektionsläufe starten. Die Zwischenzustände werden an die anderen Knoten verteilt, welche die Fehlerinjektionen durchführen. Abbildung 17

### 3. Effiziente Experimentdurchführung

zeigt, wie vom ausgezeichneten Knoten 0 zu bestimmten Zeitpunkten berechnete Zwischenzustände an die Knoten 1 bis N-1 übertragen werden. Hat einer der Knoten 1 bis N-1 eine einzelne Fehlerinjektion durchgeführt, kann er einen neuen Zwischenzustand beim Knoten 0 anfordern, bis alle durchzuführenden Experimente simuliert sind. Dadurch ergibt sich eine automatische Lastverteilung („Farming“).

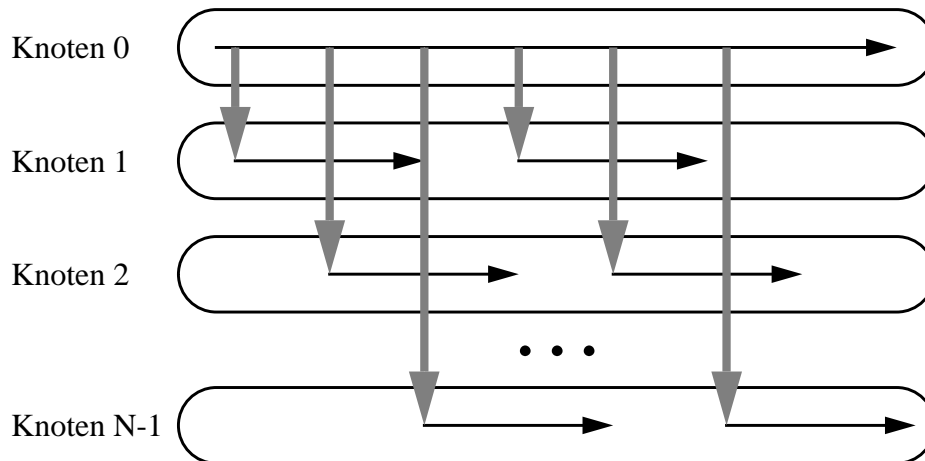


Abb. 17: Master-Knoten berechnet Golden-Run

2. Jeder Knoten berechnet den Golden-Run mit Zwischenzuständen, die der Knoten selbst zur Fehlerinjektion verwendet.

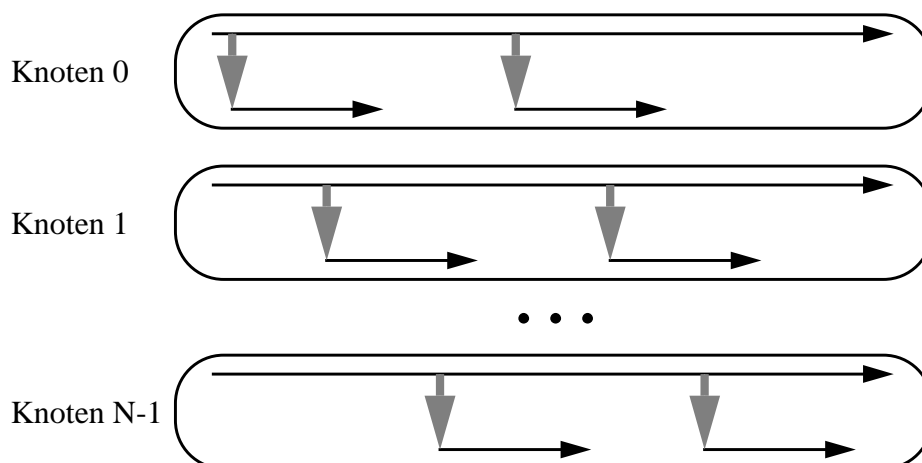


Abb. 18: Jeder Knoten berechnet Golden-Run für sich

#### 3.5.2 Bewertung des Verfahrens

Im ersten Fall entstehen zusätzliche Kosten durch das Übertragen der Zwischenzustände zu anderen Knoten. Bei  $M$  zu übertragenden Zwischenzuständen und einer jeweiligen Übertragungsdauer  $T_1$  Zeiteinheiten (abhängig von der Größe der Zwischenzustände und vom zur Verfügung stehenden Übertragungsmedium) beträgt der Overhead  $MT_1$  Zeiteinheiten. Unter Umständen reicht es aus, nur Teile eines Zwischenzustandes zu übertragen (siehe Abschnitt 3.3 und 3.4).

Im Falle, daß jeder der  $N$  Knoten den Golden-Run selbst durchführt, beträgt der Mehraufwand  $(N - 1) T_2$  ( $T_2$ : Zeit, welche die Berechnung eines Golden-Runs benötigt), da der Golden-Run  $N$ -mal berechnet, aber nur einmal benötigt wird.

Je nach Größe der Werte  $N$ ,  $M$ ,  $T_1$  und  $T_2$  sollte Verfahren 1 bzw. 2 ausgewählt werden. Eigene Versuche haben gezeigt, daß beide Verfahren eine Beschleunigung um einen Faktor nahe  $N$  bringen können.

## 3.6 Vergleich mit Golden-Run

### 3.6.1 Prinzipielles Verfahren

Wie die Erfahrung zeigt (siehe Kapitel 6.3), hat die Fehlerinjektion in vielen Experimenten (z.B. bei über 50% der Experimente mit Stuck-At-Fehlern) keinerlei Auswirkungen, da der injizierte Fehler sofort maskiert wird und somit nicht zur Auswirkung gelangt. Ebenso weiß man, daß Fehler existieren, deren Auswirkungen nach relativ kurzer Zeit wieder verschwinden. Der Fall, daß das untersuchte System durch einen injizierten Fehler langfristig gestört wird, ist eher die Ausnahme.

Wenn das zu simulierende Modell deterministisch arbeitet, d.h. der Zustand des Systems eindeutig seinen weiteren Ablauf bestimmt (im Gegensatz zu realen Systemen ist dies bei Simulationen die Regel), kann eine Fehlersimulation folgendermaßen beschleunigt werden:

Wenn ein Testlauf synchron zu einem Golden-Run ausgeführt wird, besteht die Möglichkeit, nach jedem Simulationsschritt die Zustände der beiden Simulationen zu vergleichen. Wenn sie übereinstimmen, kann der Testlauf abgebrochen werden, da dann der Ablauf des fehlerlosen Systems mit dem des mit Fehlern versehenen Systems auf Dauer übereinstimmen wird. Alle Fragen bezüglich des Verhaltens des Systems können dann vor Ablauf des eigentlichen Beobachtungsintervalls durch eine Inspektion des Golden-Runs beantwortet werden.

Ein Nachteil dieses Verfahrens ist der doppelte Speicher- und bis zum Abbruch der Simulation doppelte Rechenzeitbedarf. Ebenfalls nachteilig können sich die notwendigen Vergleiche auswirken und den Zeitvorteil, den ein vorzeitiger Abbruch einer Simulation bewirkt, wieder aufzehren. Die Vergleiche sollten daher so schnell wie möglich ausgeführt werden. Dazu wurden zwei Verfahren entwickelt, die in den folgenden Absätzen genauer beschrieben werden (Abschnitt 3.6.2 und 3.6.3). Ein Vergleich der beiden Methoden sowie Messungen bezüglich ihres Nutzens und ihrer Kosten enthält der Abschnitt 3.6.4.

### 3.6.2 Zustandsvergleich durch Signaturen

Eine Möglichkeit, schnell vergleichen zu können, besteht darin, daß der Simulator jedem Zustand eine Signatur zuordnet, die sich aus allen Teilzuständen berechnet. Verglichen werden dann zunächst nur die Signaturen vom Golden-Run und vom synchron simulierten Testlauf. Nur wenn die Signaturen übereinstimmen, werden die Zustände selber miteinander verglichen. Damit eine Beschleunigung der Experimente erreicht werden kann, sollten die Signaturen bei diesem Verfahren die Eigenschaften haben, daß sie schnell berechnet werden können und daß zwei Signaturen i.a. nur dann übereinstimmen, wenn sie auch gleiche Zustände repräsentieren.

Besonders schnell lassen sich die Signaturen  $S$  berechnen, wenn sie sich auf einfache Weise beim Übergang des Systems von einem Zustand  $Z = (z_1, z_2, \dots, z_i, \dots, z_N)$  in einen Zustand  $Z' = (z_1, z_2, \dots, z'_i, \dots, z_N)$  als  $S(Z') = f(S(Z), z_i, z'_i)$  berechnen lassen; das heißt, wenn sich die Signatur des neuen Zustandes aus der Signatur des alten Zustandes ( $S(Z)$ ) und seiner Veränderung ( $z_i$  nach  $z'_i$ ) ergibt.

### 3. Effiziente Experimentdurchführung

---

Beispiele für einfach zu berechnende Signaturen sind:

- Die Summe über alle Teilzustände<sup>1</sup>:  $S(Z) = z_1 + z_2 + \dots + z_N$ . Für  $S(Z')$  ergibt sich dann:  $S(Z') = S(Z) - z_i + z'_i$ .
- Das Exklusiv-Oder über alle Teilzustände:  $S(Z) = z_1 \text{ XOR } z_2 \text{ XOR } \dots \text{ XOR } z_N$ . Für  $S(Z')$  ergibt sich dann:  $S(Z') = S(Z) \text{ XOR } z_i \text{ XOR } z'_i$ .

Da in vielen Simulationssprachen der Anfangszustand als  $Z_0 = (0, 0, \dots, 0)$  definiert ist (z.B. auch in den häufig verwendeten Sprachen VHDL und VERILOG), ist auch die Berechnung der ersten Signatur in den oben genannten Beispielen einfach:  $S(0, 0, \dots, 0) = 0$ .

#### 3.6.3 Inkrementeller Zustandsvergleich

Eine andere Möglichkeit, nach jedem Simulationsschritt schnell zwei Zustände auf Gleichheit zu überprüfen, besteht darin, daß ein Zähler mitgeführt wird, der die Anzahl der verschiedenen Teilzustände anzeigt. Enthält der Zähler den Wert 0, sind beide Zustände identisch. Damit der Zähler ständig den korrekten Wert anzeigt, muß er bei jeder Änderung des Zustandes des Golden-Run sowie des Testlaufes aktualisiert werden. Dies kann nach folgendem Algorithmus geschehen:

Bevor die Fehlerinjektion stattfindet, ist der Wert des Zählers 0.

Wenn der Zustand des Golden-Run sich von  $G = (g_0, g_1, \dots, g_i, \dots, g_N)$  auf den Wert  $G' = (g_0, g_1, \dots, g'_i, \dots, g_N)$  ändert ( $g_i \neq g'_i$ ), während sich das Testsystem im Zustand  $T = (t_0, t_1, \dots, t_i, \dots, t_N)$  befindet,

- erhöht sich der Zähler um 1, wenn  $g_i = t_i$  und  $g'_i \neq t_i$ ;
- erniedrigt sich der Zähler um 1, wenn  $g_i \neq t_i$  und  $g'_i = t_i$ .

Wenn der Zustand des Testsystems sich von  $T = (t_0, t_1, \dots, t_i, \dots, t_N)$  auf den Wert  $T' = (t_0, t_1, \dots, t'_i, \dots, t_N)$  ändert ( $t_i \neq t'_i$ ), während sich das Vergleichssystem im Zustand  $G = (g_0, g_1, \dots, g_i, \dots, g_N)$  befindet,

- erhöht sich der Zähler um 1, wenn  $g_i = t_i$  und  $g_i \neq t'_i$ ;
- erniedrigt sich der Zähler um 1, wenn  $g_i \neq t_i$  und  $g_i = t'_i$ .

Erreicht der Zähler den Wert 0, sind beide Zustände identisch. Ändern sich mehrere Teilzustände zum gleichen, simulierten Zeitpunkt, ist die Abfrage, ob der Zähler den Wert 0 erreicht hat, erst nach der Durchführung aller Zustandsänderungen durchzuführen.

---

1. Teilzustände können z.B. das Potential eines Signals oder der Inhalt eines Registers oder einer Speicherzelle des Hauptspeichers sein.

### 3.6.4 Bewertung der Verfahren

Durch die Anwendung dieser Verfahren bei durchgeführten Experimenten (Beschreibung siehe Kapitel 6) konnte die Durchführung der Simulation zum Teil deutlich beschleunigt werden. Tabelle 2 zeigt die gemessenen Beschleunigungsfaktoren für verschiedene Experimente. Für die Signaturberechnung wurde die Summenbildung über alle Teilzustände verwendet.

**Tabelle 2: Beschleunigungsfaktoren durch Vergleich mit Golden-Run**

Art des Versuches	maximale Beobachtungsdauer	Beschleunigungsfaktor Zustandsvergleich durch Signaturen	Beschleunigungsfaktor Inkrementeller Zustandsvergleich
Stuck-At-Fehler	2ms	14.4	18.8
	4ms	26.3	36.1
	6ms	39.4	47.4
Pin-Level-Fehler	2ms	8.8	10.7
	4ms	11.0	15.0
	6ms	19.4	23.7
Bit-Flip-Fehler	2ms	1.3	1.2
	4ms	2.5	2.4
	6ms	4.4	3.5

Zu sehen ist, daß die Fehlerinjektionsexperimente bei Fehlern, die selten Auswirkungen zeigen oder deren Auswirkungen schnell abklingen (hier: Stuck-At- und Pin-Level-Fehler), durch die beschriebenen Verfahren deutlich beschleunigt werden können. Die Verfahren zeigen eine geringere Effektivität bei Fehlern, deren Recovery-Zeit größer ist (hier: Bit-Flip-Fehler). Je größer jedoch das Beobachtungsintervall wird, desto größer wird der Beschleunigungsfaktor. Im Falle der Stuck-At- und Pin-Level-Fehler ist die Zunahme sogar nahezu proportional zur Beobachtungsdauer. Diese Verfahren werden also um so attraktiver, je länger einzelne Fehler beobachtet werden sollen.

Genauere Untersuchungen an Beispielen (siehe Kapitel 6) haben gezeigt, daß die Summe (32-Bit-Arithmetik) aller Teilzustände als Signatur sehr gut geeignet ist, um zwei Zustände schnell grob zu vergleichen. Wenn zwei auf die beschriebene Weise berechnete Signaturen übereinstimmen, kann mit über 99,99% Wahrscheinlichkeit davon ausgegangen werden, daß auch die damit repräsentierten Zustände gleich sind.

Wie an den Berechnungsalgorithmen zu erkennen ist, ist das Verfahren des inkrementellen Vergleichens während der Vergleichsphase aufwendiger und damit langsamer als das Verfahren des Signatur-Vergleichs. Wenn jedoch der Signaturvergleich zufälligerweise während der Vergleichsphase oder nach einer erfolgreichen Recovery des Systems eine mögliche Übereinstimmung der beiden Zustände anzeigt, müssen bei diesem Verfahren die Zustände nochmals gründlich und damit zeitaufwendig miteinander verglichen werden.

Für eine Fehlersimulation sollte daher das Verfahren des Signatur-Vergleiches benutzt werden, wenn wenige Fehlerinjektionen durchgeführt werden sollen, jeder injizierte Fehler aber vermutlich eine lange Beobachtungsdauer erfordert (z.B. Bit-Flip-Fehler). Inkrementelles Vergleichen

### 3. Effiziente Experimentdurchführung

---

lohnt sich dagegen besonders bei vielen durchzuführenden Injektionen mit - z.B. aufgrund schneller Recovery-Verfahren - kurzen Beobachtungsintervallen (z.B. Stuck-At- und Pin-Level-Fehler) sowie bei besonders großen Modellen.

Insgesamt bedeuten diese Ergebnisse für die Praxis, daß nahezu beliebig große Zustände ohne großen Zeitverlust miteinander verglichen werden können, da der Aufwand, zwei Signaturen zu überprüfen bzw. einen inkrementellen Vergleich durchzuführen, unabhängig von der Größe des Modells ist.

## 3.7 Dynamisches Wechseln von Modellen

Im allgemeinen ist die Komplexität eines Modells ausschlaggebend für den Aufwand, es zu simulieren. Um die Simulation zu beschleunigen, kann man deshalb versuchen, das Modell zu vereinfachen. Leider ist ein einfaches Modell nicht mehr so aussagekräftig wie ein detailliertes. Da bei Fehlerinjektionsexperimenten jedoch nur Aussagen über das System nach einer Fehlerinjektion gewonnen werden sollen, ist es sinnvoll, vor einer Fehlerinjektion ein möglichst einfach zu simulierendes Modell des Systems, nach der Fehlerinjektion ein detailliertes, gut beobachtbares Modell zu verwenden. Zum Fehlerinjektionszeitpunkt muß das Modell dynamisch durch den Simulator gewechselt werden. Dieses Verfahren wurde in [Yang92a], [Yang92b] und [Riecken95] beschrieben (einen Überblick gibt [Iyer94]). Es läßt sich gut in die in Abschnitt 3.3 („Simulation mit Hilfe des Zielsystems“) bzw. 3.4 („Multi-Threaded Fault-Injection“) beschriebenen Verfahren integrieren. Die Schätzungen für den möglichen Beschleunigungsfaktor liegen – je nach verwendetem Modell – zwischen 5 und 25 (manuelle Optimierung [Riecken95]) und zwischen 2 und 8 (automatische Optimierung [Tschäche96a]).

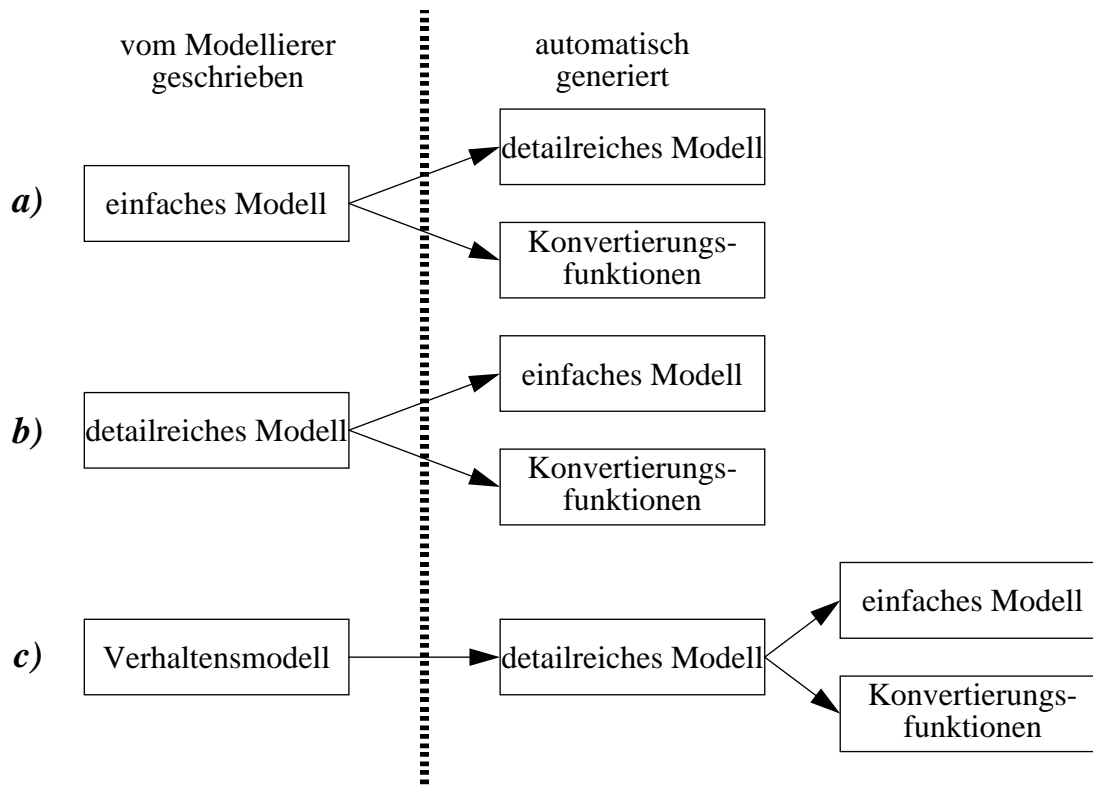
### 3.7.1 Probleme beim dynamischen Wechseln von Modellen

Dieses Verfahren wird jedoch - trotz sehr guter Leistungssteigerung - selten eingesetzt. Der Grund dafür liegt darin, daß zwei Modelle und ein geeigneter Umschaltmechanismus zur Simulation erforderlich sind. Es müssen zwei Modelle sowie Zustandskonvertierungsroutinen programmiert werden, die dafür sorgen, daß der Systemzustand, den das einfache Modell zum Fehlerzeitpunkt repräsentiert, in das detailliertere Modell übertragen werden kann. Diese Konvertierungsroutinen sind zum Teil sehr aufwendig, da sie Zustandsdaten rekonstruieren müssen, die im einfachen Modell aus Leistungsgründen weggelassen wurden. Zum Beispiel wird man ein Multiplikationswerk einer CPU nur durch eine einzige, nicht zeitbehaftete Komponente modellieren. Das detailliertere Modell muß jedoch eine Multiplikations-Pipeline nachbilden, deren einzelne Stufen zu jedem Zeitpunkt Daten enthalten. Diese sind durch die Konvertierungsfunktionen so zu rekonstruieren, daß das detailliertere Modell in einen Zustand versetzt wird, in dem es auch gewesen wäre, wenn es statt des einfachen Modells von Beginn der Simulation an verwendet worden wäre.

Sinnvoll ist daher der Versuch, das zweite Modell und die dazu passenden Konvertierungsfunktionen automatisch aus dem ersten Modell generieren zu lassen. Mehrere Ansätze sind denkbar. Die erste Möglichkeit besteht darin, den Systementwickler ein einfaches Modell schreiben zu lassen, aus dem dann durch ein geeignetes Werkzeug ein detailreiches Modell und die Konvertierungsroutinen erzeugt werden (Abbildung 19a). Umgekehrt könnte es auch möglich sein, ein detailliertes Modell aufbauen zu lassen, das dann automatisch vereinfacht wird (Abbildung 19b).

Diese beiden Ansätze haben jedoch ihre Schwächen. Wenn Systementwickler einfache Modelle erstellen, werden im allgemeinen keine Zustandsautomaten verwendet, da diese im Vergleich zu Verhaltensbeschreibungen mühsam und fehlerträchtig zu programmieren sind. Zustände von Systemen zu extrahieren, von denen eine einfache Verhaltensbeschreibung vorliegt, ist jedoch schwierig, da die Zustände nicht nur in Form von Variablen der einzelnen Prozesse vorliegen. Statt dessen ist diese Information auf die Variablen, die Programmzähler, den Stack, die Signalewerte und ähnliche Zustände verteilt. Die andere Möglichkeit, daß Systementwickler detaillierte Modelle schreiben, zu denen einfachere Modelle automatisch erzeugt werden, entfällt ebenfalls, da die manuelle Entwicklung größerer Modelle auf z.B. Gatterebene praktisch unmöglich ist.

Genauer untersucht wurde daher eine dritte Alternative. Der Systemmodellierer schreibt Modelle auf höherer Abstraktionsebene, die automatisch in Modelle niedrigerer Abstraktionsebene aber höherem Detaillierungsgrad umgewandelt werden. VHDL Modelle auf Verhaltensebene können z.B. durch kommerzielle VHDL-Compiler (z.B. SYNOPSIS-Compiler) in Modelle auf Gatterebene transformiert werden. Es müssen bei diesem Schritt keine Konvertierungsfunktionen erzeugt werden. Er kann Teil des Modellaufbaus sein (siehe Abschnitt 2.5). Dieses so erzeugte, detailreiche Modell wird dann verwendet, um in einem zweiten Schritt ein vereinfachtes Modell und Zustandskonvertierungsroutinen zu generieren (Abbildung 19c).



**Abb. 19: Automatische Generierung einfacher und detailreicher Modelle**

Der Vorteil dieser dritten Möglichkeit liegt darin, daß das detailreiche Modell – bedingt durch seine automatische Generierung aus dem vom Designer vorgegebenen Verhaltensmodell – einer gewissen Systematik entspricht. Deshalb können leichter ein einfaches Modell sowie passende Konvertierungsfunktionen daraus generiert werden.

### 3. Effiziente Experimentdurchführung

Diese Konvertierungen eines Modells sind nur sehr mühsam manuell durchführbar. In Modellen mit mehreren Tausend einzelner Komponenten ist eine derartige Arbeit für den Menschen in der Praxis nicht mehr in kurzer Zeit fehlerfrei durchführbar. Die Arbeit muß daher weitgehend automatisch ausgeführt werden. Im folgenden soll gezeigt werden, wie eine derartige Zusammenfassung von Komponenten vom Rechner erledigt werden kann. Eine genaue Spezifikation und Implementierung eines solchen Werkzeugs für die Modellierungssprache VHDL findet sich in [Tschäche96a] und [Tschäche96b].

#### 3.7.2 Automatische Generierung schneller simulierbarer Modelle

Wie einfache Versuche mit VHDL-Modellen auf Gatterebene zeigen, wird über 99% der Simulationszeit dafür benötigt, zwischen den einzelnen Prozessen der Komponenten hin- und herzuschalten. Die Berechnungen der Ausgaben und der Zustände der einzelnen Gatter, Register usw. selbst verbrauchen dagegen unter 1% der Rechenleistung. Ähnliches gilt für den Speicherplatzbedarf. Die Prozeß- und Signalverwaltung nimmt mehr als 90% des für die Simulation benötigten Speicherplatzes ein. Daher verspricht eine Methode, die mehrere Prozesse miteinander vereint, eine Beschleunigung der Simulation. Werden Prozesse miteinander verschmolzen, sind im allgemeinen auch die dazwischenliegenden Signale („koordinierende Variablen“) überflüssig und können entfernt oder in prozeßinterne, einfache Variablen umgewandelt werden. Selbstverständlich sollten auch Fehlersignale (siehe Abschnitt 2.6 „VHDL-Erweiterung“) aus dem Modell entfernt werden, da sie in der Zeit vor der Fehlerinjektion nicht benötigt werden. Ein Zugriff auf ein Fehlersignal kann statt dessen durch den konstanten Wert FALSE ersetzt werden.

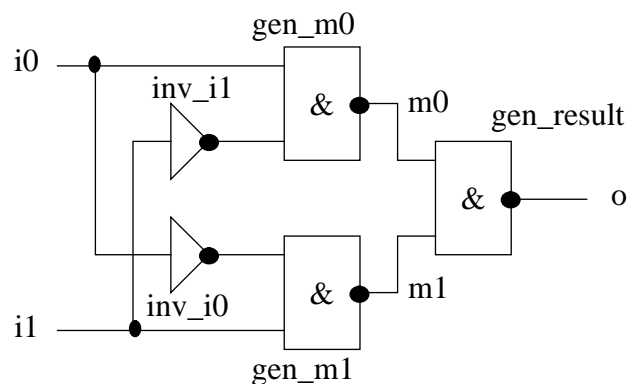
#### Verschmelzung von kombinatorischen Blöcken

*Definition „kombinatorischer Block“:*

*Bei einem „kombinatorischen Block“ (Schaltnetz) hängt der Wert der Ausgangssignale nur vom Wert der Eingangswerte ab.*

Ein kombinatorischer Block enthält demnach keinen internen Zustand. In einer Beschreibung eines Systems auf Gatterebene kommen viele Komponenten vor, die derartige kombinatorische Blöcke darstellen. Dazu gehören zum Beispiel alle AND-, OR-, NAND-, NOR-, XOR- und NOT-Gatter, alle Multiplexer und Demultiplexer, alle Encoder und Decoder. Eine Ausnahme bilden i.a. nur speichernde Elemente wie z.B. die Register, Latches und Zähler.

Im folgenden wird am Beispiel eines XOR-Gatters (siehe Abbildung 20) gezeigt, wie mehrere kombinatorische Blöcke zu einem einzelnen zusammengefaßt werden können.



**Abb. 20: XOR-Gatter aus einzelnen Invertiern und NAND-Gattern**

Die VHDL-Beschreibung auf Gatterebene verwendet für diese Darstellung des XOR-Gatters zwei Inverter und drei NAND-Gatter (siehe Abbildung 21 und 22). Es sind fünf Prozesse, sowie vier interne Signale notwendig.

```
ENTITY xor2 IS
    PORT(
        i0, i1: IN bit;
        o: OUT bit);
END xor2;
```

**Abb. 21: VHDL-Entity-Beschreibung eines XOR-Gatters**

```
ARCHITECTURE gate OF xor2 IS
    SIGNAL ii0, ii1: bit;
    SIGNAL m0, m1: bit;
BEGIN
    gen_result: PROCESS(m0, m1) BEGIN
        o <= m0 NAND m1;
    END PROCESS;
    gen_m0: PROCESS(i0, ii1) BEGIN
        m0 <= i0 NAND ii1;
    END PROCESS;
    gen_m1: PROCESS(i1, ii0) BEGIN
        m1 <= i1 NAND ii0;
    END PROCESS;
    inv_i0: PROCESS(i0) BEGIN
        ii0 <= NOT i0;
    END PROCESS;
    inv_i1: PROCESS(i1) BEGIN
        ii1 <= NOT i1;
    END PROCESS;
END gate;
```

**Abb. 22: VHDL-Code für XOR-Gatter (nicht optimiert)**

In der Praxis wird eine strukturelle Beschreibung einer Komponente nicht – wie in Abbildung 22 dargestellt – mehrere Prozesse, sondern Instanzen anderer Subkomponenten beinhalten, die wiederum Prozesse enthalten. Diese Prozesse der Subkomponenten lassen sich jedoch in die übergeordnete Komponente hineinkopieren, so daß eine Darstellung wie in Abbildung 22 entsteht. Dies läßt sich durch geeignete Transformationen, die jedes gute Synthesewerkzeug bietet, automatisch durchführen („Model Flattening“).

Die optimierte Version dieses XOR-Gatters (Abbildung 23) besteht nur noch aus einem einzigen Prozeß. Da der Zugriff auf die Werte `ii0`, `ii1`, `m0` und `m1` nun nicht mehr zwischen verschiedenen Prozessen koordiniert werden muß, können statt der Signale prozeßinterne Variablen verwendet werden. Eigene Messungen bestätigen, daß diese Modifikation der Beschreibung des XOR-Gatters eine Beschleunigung der Simulation dieser Komponente um etwa das Fünffache erreicht.

### 3. Effiziente Experimentdurchführung

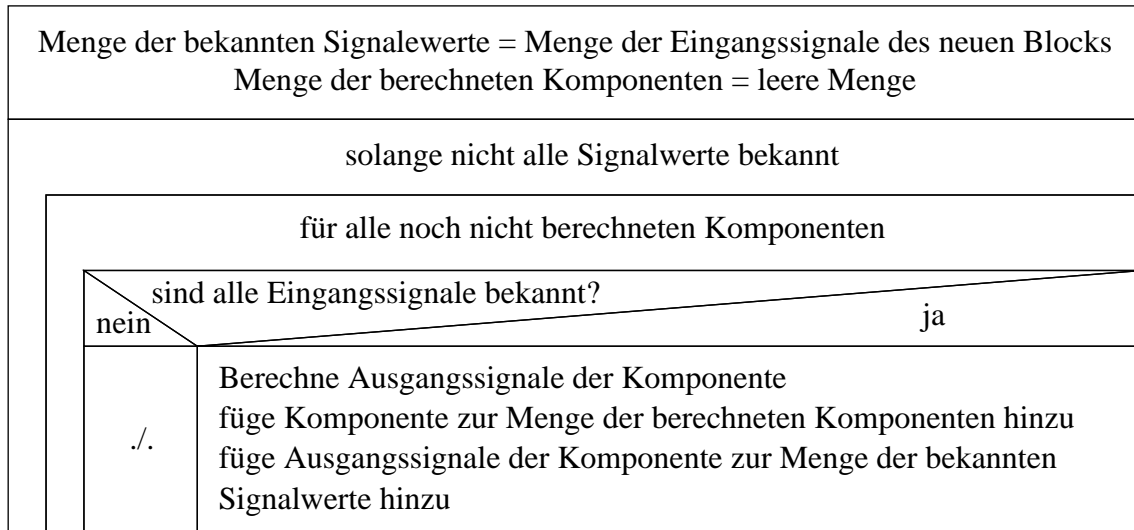
---

```
ARCHITECTURE optimized OF xor2 IS
BEGIN
  PROCESS(i0, i1)
    VARIABLE ii0, ii1: bit;
    VARIABLE m0, m1: bit;
  BEGIN
    ii0 = NOT i0;
    ii1 = NOT i1;
    m0 = i0 NAND ii1;
    m1 = i1 NAND ii0;
    o <= m0 NAND m1;
  END PROCESS;
END optimized;
```

**Abb. 23: VHDL-Code für XOR-Gatter (Prozeß-optimiert)**

Allgemein können kombinatorische Blöcke zu einem einzelnen kombinatorischen Block zusammengefaßt werden, wenn das Netz, das durch die Blöcke und die verbindenden Signale gebildet wird, keine Rückkopplungen enthält. Bei der Verschmelzung der einzelnen Blöcke muß jedoch auf die Berechnungsreihenfolge der einzelnen Prozesse geachtet werden. Beispielsweise muß im XOR von Abbildung 20 erst der Wert des Ausgangs von Gatter *inv\_i0* (*ii0*) bekannt sein, bevor der Ausgangswert von Gatter *gen\_m1* (*m1*) berechnet werden kann. Eine mögliche Berechnungsreihenfolge wäre *inv\_i0*, *inv\_i1*, *gen\_m0*, *gen\_m1*, *gen\_result*. Sind in einer Beschreibung eines Systems in VHDL mehrere Prozesse über Signale miteinander verbunden, sorgt der VHDL-Simulator für die richtige Reihenfolge bei der Abarbeitung der Prozesse oder notfalls für die mehrfache Ausführung von Prozessen. Werden mehrere Prozesse zu einem einzelnen zusammengefaßt, muß die Reihenfolge der Berechnungsschritte jedoch stimmen, da die Anweisungen innerhalb eines Prozesses nicht vom VHDL-Simulator umgeordnet oder mehrfach ausgeführt werden.

Das heißt, daß der neugebildete Prozeß zunächst die Ausgangswerte von Komponenten berechnen muß, deren Eingangssignale eine Teilmenge der Eingangssignale der neuen, großen Komponente sind. Dies ist nach der Definition eines kombinatorischen Blockes möglich. Darauf folgend müssen die Ausgangssignale aller Komponenten berechnet werden, deren Eingangssignale durch die gerade berechneten Komponenten vollständig bekannt geworden sind. Dieser Vorgang wird solange wiederholt, bis alle Ausgangssignale berechnet sind. Da keine Schleifen vorkommen dürfen, ist es möglich, auf diese Weise alle Prozesse in einer richtigen Reihenfolge in den neuen Prozeß zu übernehmen. Abbildung 24 zeigt ein Struktogramm dieses Algorithmus.



**Abb. 24: Struktogramm: Ermittlung der Berechnungsreihenfolge**

#### Verschmelzung von Registern, Latches und Zählern

In der Praxis werden vielfach mehrere Register, Latches oder Zähler zu größeren Einheiten miteinander verbunden. In modernen Prozessoren werden z.B. häufig 32 oder 64 einzelne 1-Bit-Register zu einem Wort zusammengefaßt. Derartige größere Gruppen von Registern (bzw. Latches oder Zählern) werden vom gleichen Signal getaktet. Eine solche Gruppe von Prozessen kann dann leicht in einen einzigen Prozeß umgewandelt werden, wie folgendes Beispiel in den Abbildungen 25 bis 27 zeigt:

```

ENTITY word IS
    PORT ( i: IN bit(31 DOWNT0 0);
           clk: IN bit;
           o: OUT bit(31 DOWNT0 0)
    );
END word;
    
```

**Abb. 25: VHDL-Entity-Beschreibung eines 32-Bit-Registers**

### 3. Effiziente Experimentdurchführung

---

```
ARCHITECTURE gate OF word IS
BEGIN
  b0: PROCESS(clk) BEGIN
    IF clk = '1' THEN o(0) <= i(0); END IF;
  END PROCESS;
  b1: PROCESS(clk) BEGIN
    IF clk = '1' THEN o(1) <= i(1); END IF;
  END PROCESS;
  ...
  b31: PROCESS(clk) BEGIN
    IF clk = '1' THEN o(31) <= i(31); END IF;
  END PROCESS;
END gate;
```

**Abb. 26: VHDL-Beschreibung eines 32-Bit-Registers (Gatterebene, nicht optimiert)**

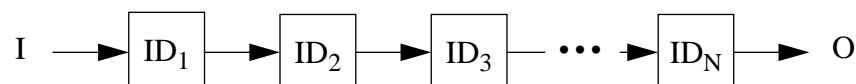
```
ARCHITECTURE optimized OF word IS
BEGIN
  b0to31: PROCESS(clk) BEGIN
    IF clk = '1' THEN o(0) <= i(0); END IF;
    IF clk = '1' THEN o(1) <= i(1); END IF;
    ...
    IF clk = '1' THEN o(31) <= i(31); END IF;
  END PROCESS;
END optimized;
```

**Abb. 27: VHDL-Beschreibung eines 32-Bit-Registers (Gatterebene, Prozeß-optimiert)**

Ein optimierender VHDL-Compiler kann die Beschreibung in Abbildung 27 weiter optimieren und z.B. die mehrfach angegebene IF-Anweisung nur einfach ausführen lassen. Derartige Optimierungen gehören inzwischen zum Standard beim Compiler-Bau (siehe z.B. [Aho89]). Die Prozeßverschmelzung kann daher noch weitere, zusätzliche Optimierungsmöglichkeiten bieten.

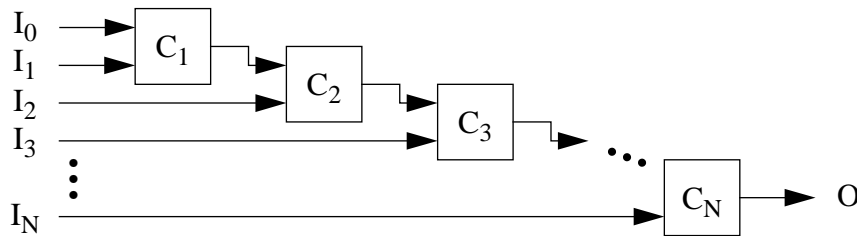
#### 3.7.3 Bewertung

Die maximal mögliche Beschleunigung dieses Verfahrens erhält man, wenn man eine Reihe von Identitäts-Gattern ( $ID_1$  bis  $ID_N$ ) hintereinanderschaltet (Abbildung 28). Diese lassen sich, da sie alle kombinatorische Blöcke sind und keine Rückkopplungen enthalten, zu einem einzelnen Prozeß zusammenfassen. Messungen haben gezeigt, daß im Falle von  $N$  Gattern die Beschleunigung, die mit diesem Verfahren zu erzielen ist, auch fast  $N$  erreicht (mit dem MODELTECH-VHDL-Compiler und -Simulator beträgt die Beschleunigung für große  $N$  ca.  $0.996 \cdot N$ ).



**Abb. 28: Best-Case bei der Verschmelzung von Prozessen**

Da im Falle praxisorientierter Modelle von getakteten Systemen jedoch nur mit ca. zehn Gattern hintereinander zu rechnen ist, kann das Verfahren in der Praxis die Simulation um höchstens einen Faktor in dieser Größenordnung beschleunigen. Ein weiteres Problem dieses Verfahrens wird anhand eines Beispiels in Abbildung 29 erläutert.



**Abb. 29: Worst-Case bei der Verschmelzung von Prozessen**

Ändert sich in diesem Beispiel, bedingt durch den angelegten Stimulus, nur der Wert des Eingangssignals  $I_N$ , so muß im Original-Gattermodell nur das Ausgangssignal der Komponente  $C_N$  neu bestimmt werden. Die Prozesse der Komponenten  $C_1$  bis  $C_{N-1}$  brauchen nicht „aufgeweckt“ zu werden, da sich ihre Sensitivsignale nicht ändern. Sind dagegen alle Prozesse der Komponenten  $C_1$  bis  $C_N$  zu einem einzelnen Prozeß zusammengefaßt, kann nur dieser Prozeß als ganzes ausgeführt, müssen alle Zwischenwerte neu berechnet werden. In diesem Fall wird keine Beschleunigung zu beobachten sein, sondern – im Gegenteil – eine langsamere Simulationsgeschwindigkeit.

Daher sollten nur solche Prozesse miteinander verschmolzen werden, die bei den vorkommenden Stimuli meistens gleichzeitig aktiv bzw. gleichzeitig nicht aktiv sind. Da dies sowohl von der Struktur des verwendeten Modells als auch vom Stimulus abhängig ist, kann diese Auswahl nicht automatisch getroffen werden, da der Stimulus zur Zeit der Modellgenerierung i.a. nicht bekannt ist. Es muß also vom Benutzer vorgegeben werden, welche Komponenten miteinander verschmolzen werden sollen.

Untersuchungen [Tschäche96a] haben anhand verschiedener größerer Beispiele aus der Praxis gezeigt, daß ein erfahrener Benutzer ohne großen manuellen Aufwand durch Einsatz dieses Verfahrens seine Simulationen um einen Faktor zwischen 2 und 8 beschleunigen kann.

## 3.8 Irrelevante Fehler in Registern

### 3.8.1 Beschreibung des Verfahrens

Bei vielen Fehlerinjektionsexperimenten ist zu beobachten, daß injizierte Fehler nach außen keinerlei beobachtbare Auswirkungen zeigen. Genauere Untersuchungen beweisen jedoch, daß sich in einigen dieser Fälle sehr wohl intern der Zustand des fehlerhaften Systems über längere Zeit vom fehlerfreien System unterscheidet. Ein Beispiel hierfür ist eine Fehlerinjektion in ein Register einer CPU. Wird dieses spezielle Register vom Programm zur Zeit der Fehlerinjektion nicht verwendet, können die im Register enthaltenen, fehlerhaften Daten keinen Einfluß auf den Fortgang der Rechnung des Systems nehmen. Das laufende Programm wird weiterhin zu einem korrekten Ergebnis kommen, obwohl der interne Zustand des Systems über eine gewisse Zeit fehlerhaft ist. Fehlerinjektionen in Register bzw. in Speicherzellen sind daher nur vor einem Lesezugriff sinnvoll. Geschehen sie ohne vorigen Lesezugriff vor einem Schreibvorgang, können sie keine Auswirkungen haben, da der Fehler durch das Überschreiben wieder korrigiert wird (siehe Abbildung 30).

### 3. Effiziente Experimentdurchführung

```
PROCEDURE proc IS
  VARIABLE x;
BEGIN
  ...;           <- ein Fehler in x hat (noch) keine Auswirkungen
  x := f(y);     <- erste Zuweisung an x
  ...;           <- ein Fehler in x hat i.a. Auswirkungen
  z := g(x);     <- letzte Verwendung von x
  ...;           <- ein Fehler in x hat keine Auswirkungen (mehr)
END;
```

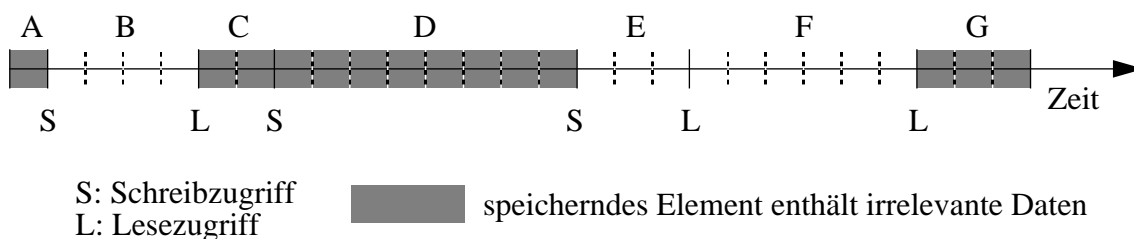
**Abb. 30: Lebenszyklen von Zuständen**

Das Beispiel zeigt, daß Manipulationen am Zustand des Systems (z.B. durch Bit-Fehler in Registern oder im Hauptspeicher) nur zu bestimmten Zeiten Auswirkungen zeigen werden. Weder vor der ersten Zuweisung an eine Variable noch nach ihrer letzten Verwendung wird sich eine Wertänderung dieser Variablen durch ein fehlerhaftes Systemverhalten bemerkbar machen. Register- oder Speicherfehler, die zwischen einem Schreib- und einem Lesevorgang auftreten, können dagegen zu Ausfällen des Gesamtsystems führen.

Unterteilt man die gesamte Zeit einer Simulation in einzelne Intervalle, die durch Schreib- bzw. Lesevorgänge eines bestimmten Registers bzw. einer bestimmten Speicherzelle begrenzt sind, ergeben sich für jedes Intervall die folgenden zwei Möglichkeiten:

- Intervall endet mit Lesezugriff
- Intervall endet mit Schreibzugriff

Endet ein Intervall mit einem Schreibzugriff auf das entsprechende speichernde Element (oder mit dem Ende der Benutzeranwendung), ist das Ergebnis einer Fehlerinjektion in dieses Element innerhalb des Intervalls als „auswirkunglos“ vorhersagbar. Der Fehler wird in diesem Fall durch den nachfolgenden Schreibvorgang korrigiert. Abbildung 31 verdeutlicht die Bedeutung dieser Intervalle.



**Abb. 31: Intervalle, in denen ein speicherndes Element keine relevanten Daten enthält**

Im Beispiel wird in den Intervallen A, C, D und G eine Fehlerinjektion in den zum Diagramm gehörenden Teilzustand (Register oder Speicherzelle) keine Auswirkungen haben, da der Fehler durch den Schreibzugriff am Ende des Intervalls (Intervall A, C, D) bzw. durch das Ende der Benutzeranwendung (Intervall G) keinen Einfluß auf den Fortgang der Rechnung nehmen kann. Am Ende der übrigen Intervalle wird der Teilzustand jedoch ausgelesen und weiterverarbeitet. In den Fällen ist anhand dieses Diagramms keine Aussage möglich, ob der Fehler letztendlich das Ergebnis der Rechnung beeinflusst. Dies ist dann nur durch eine genauere Untersuchung des Systems (z.B. durch eine Simulation) zu ermitteln. Sind aber die Zeitpunkte der Schreib- und

Lesezugriffe bekannt, können in diesem Beispiel ca. 50% der Fehlerinjektionsexperimente eingespart werden, da ihr Ausgang einfacher anhand der Betrachtung der Nutzungszeiträume bestimmt werden kann. Genügt für eine bestimmte Anwendung eine grobe Abschätzung der Fehlermaskierungswahrscheinlichkeit, ist es möglich, durch diese Diagramme eine untere Schranke der Maskierungswahrscheinlichkeit anzugeben (im oberen Beispiel: ca. 50%).

Durch die Analyse der Schreib- und Lesezugriffe auf die speichernden Elemente kann weiterhin auch eine Abschätzung der Fehlerlatenzzeit erreicht werden. Tritt ein Fehler in einem Register bzw. einer Speicherzelle auf, kann er frühestens beim nächsten Lesezugriff auf diese Komponente erkannt werden. Die Fehlerlatenzzeit wird im Mittel größer sein als die Hälfte der mittleren Länge der mit einem Lesezugriff abgeschlossenen Intervalle. Im obigen Beispiel haben die mit einem Lesezugriff abgeschlossenen Intervalle ( $B$ ,  $E$ ,  $F$ ) eine Länge von 4 ( $B$ ), 3 ( $E$ ) und 6 Zeiteinheiten ( $F$ ). Ihre mittlere Länge beträgt entsprechend 6.5 Zeiteinheiten. Es ist daher anzunehmen, daß ein Fehler innerhalb dieses Speicherelements im Mittel frühestens nach 3.25 Zeiteinheiten erkannt wird.

#### 3.8.2 Bestimmung der Schreib- und Lesezugriffe

Für das eben beschriebene Verfahren ist es unerlässlich, die genauen Zeiten der Schreib- und Lesezugriffe auf die Register und Speicherzellen zu bestimmen. Dieser Abschnitt versucht daher, Methoden bereitzustellen, wie diese Zeiten ermittelt werden können. Das Einzelschrittverfahren ist besonders für Software-implementierte Fehlerinjektionsverfahren interessant, da in diesen Werkzeugen meist sowieso eine Möglichkeit vorgesehen ist, eine Applikation schrittweise auszuführen, um zum Beispiel einen Fehlerinjektionszeitpunkt anzufahren oder für eine Anzahl von Takten einen Fehler zu injizieren. Der Ansatz auf Simulationsbasis ist dagegen für den Fall interessant, daß vom zu untersuchenden System ein simulierbares Modell existiert.

##### Einzelschrittverfahren

Läßt man ein Programm von einer CPU im Einzelschrittmodus abarbeiten, so wird die CPU zwischen den einzelnen Instruktionen eine Ausnahmebehandlungsroutine durchlaufen. In dieser Routine kann auf die Werte aller Register und aller Speicherzellen der Anwendung zugegriffen werden. So ist es möglich, die jeweils nächste auszuführende Instruktion zu ermitteln. Durch Dekodierung dieser Instruktion kann bestimmt werden, auf welche Register und welche Speicherzellen während des nächsten Programmschrittes lesend bzw. schreibend zugegriffen werden wird. Dies kann z.B. in einer Trace-Datei für eine spätere Weiterverarbeitung vermerkt werden.

```
add R1, R2, (R3)      INST <= (PC)
                       (R3) <= R1 + R2
                       PC <= PC + 1
```

**Abb. 32: Erforderliche Mikroinstruktionen zur Ausführung einer Addition**

Im Beispiel in Abbildung 32 wird eine CPU bei Ausführung der Addition zunächst auf den Programmzähler lesend zugreifen, um die nächste Instruktion zu bestimmen. Danach wird die CPU aus dem Cache oder dem Hauptspeicher die Instruktion holen. Nachfolgend werden Register  $R1$  und  $R2$  abgerufen und in die ALU transportiert. Register  $R3$  wird gelesen, um die Adresse zu ermitteln, wohin das Ergebnis abgelegt werden soll. Anschließend wird an diese Adresse im Speicher das Datum geschrieben und der Programmzähler erhöht (Schreibvorgang). Während

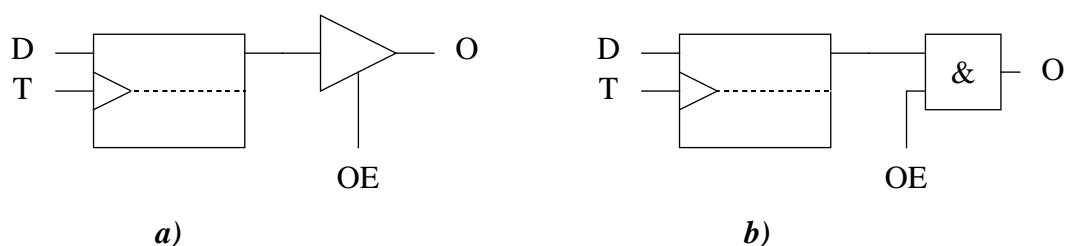
### 3. Effiziente Experimentdurchführung

der Abarbeitung dieser einen Instruktion werden also die Register  $PC$ ,  $R1$ ,  $R2$  und  $R3$  sowie die Speicherzelle mit der Adresse des  $PC$ s gelesen sowie der Programmzähler und die Speicherzelle mit der Adresse, die in  $R3$  steht, beschrieben.

Im Falle nicht-nebenläufig arbeitender CPU's ist eine Entwicklung eines derartigen Einzelschrittanalysators relativ einfach. Architekturen mit Pipeline-Strukturen können jedoch einen hohen Implementierungsaufwand erfordern, wenn die einzelnen Zugriffszeitpunkte sowohl schnell als auch genau berechnet werden sollen.

#### Simulationsverfahren

In vielen Fällen werden Register oder Latches in Verbindung mit nachfolgenden Tri-State-Treibern (Abbildung 33a) oder AND- bzw. OR-Gattern verwendet (Abbildung 33b). Ist dies auch so im Modell nachgebildet, ist es auf einfache Weise möglich, die Schreib- und Lesezeitpunkte zu bestimmen.



**Abb. 33: Register oder Latch mit nachfolgenden Gattern**

Ein Schreibvorgang vollzieht sich, wenn am Takteingang  $T$  ein „High“ anliegt (Latch) bzw. das Taktsignal von „Low“ nach „High“ wechselt (Register). Der Inhalt des Registers bzw. Latches wird ausgelesen, solange der „Output-Enable“-Eingang  $OE$  der nachfolgenden Komponente „High“ führt. Schreib- und Lesezugriffe sind in diesen Fällen häufig Vorgänge mit einer gewissen Zeitdauer. Sie definieren daher keine Zeitpunkte. Die obigen Überlegungen bezüglich der Unterteilung der Zeit in Intervalle, die durch Schreib- bzw. Lesezugriffe begrenzt sind, lassen sich jedoch leicht auf länger andauernde Zugriffe erweitern. Die Schreib- bzw. Lesephasen lassen sich im Falle einer Simulation leicht anhand der Trace-Datei eines Golden-Runs einer Simulation bestimmen. Es müssen aus der Trace-Datei nur die Signale  $T$  und  $OE$  der jeweiligen Komponenten ausgelesen werden.

#### 3.8.3 Beispiel

Um dieses Verfahren zu testen, wurde ein Einzelschrittanalysator für das MEMSY-System [DalCin94] entwickelt. Jeder Knoten des Modular Erweiterbaren Multiprozessor SYstems besteht aus einem Motorola MVME188-System mit jeweils vier Motorola-M88100 Prozessoren mit einem Systemtakt von 25 MHz, acht zugehörigen Cache- und MMU-Controllern (M88200) sowie 32 MByte Hauptspeicher.

Im ersten Beispiel wurde ein Dhrystone-Benchmark-Programm mit jeweils 500 Schleifendurchläufen untersucht. Das Testprogramm wurde mit zwei verschiedenen Compilern (GREENHILL „cc“ und GNU-C-Compiler „gcc“) und verschiedenen Compiler-Optionen (Debug-Modus „-g“ oder Laufzeitoptimierung „-O“ bzw. „-O2“) übersetzt.

Mit Hilfe des Einzelschrittanalysators wurden die Zeitpunkte der Schreib- und Lesezugriffe auf alle Register sowie alle vom Benutzerprogramm aus zugänglichen Speicherzellen registriert. Darauf aufsetzend konnte berechnet werden, in wieviel Prozent der Beobachtungszeit ein Register bzw. eine Speicherzelle relevante Daten enthält. Tabelle 3 zeigt eine Zusammenfassung der Ergebnisse.

**Tabelle 3: Mittlerer Nutzungsgrad von Speicherplätzen (Compiler-Abhängigkeit)**

Compiler	Anzahl Instruktionen	Memory			Register
		Text	Data	Stack	
cc -g	590378	24KByte 14.77% 2.00GByteInst	40KByte 3.81% 879MByteInst	4KByte 2.59% 59.7MByteInst	124Byte 49.41% 33.4MByteInst
cc	571372	24KByte 14.34% 1.88GByteInst	40KByte 3.81% 850MByteInst	4KByte 2.18% 48.6MByteInst	124Byte 49.95% 32.7MByteInst
cc -O	416445	24KByte 14.93% 1.42GByteInst	40KByte 3.81% 620MByteInst	4KByte 2.37% 38.6MByteInst	124Byte 48.05% 22.9MByteInst
gcc -g	673173	28KByte 15.19% 2.73GByteInst	40KByte 3.37% 887MByteInst	4KByte 2.45% 64.4MByteInst	124Byte 50.82% 39.2MByteInst
gcc	673173	28KByte 15.19% 2.73GByteInst	40KByte 3.37% 887MByteInst	4KByte 2.45% 64.4MByteInst	124Byte 50.82% 39.2MByteInst
gcc -O2	417176	24KByte 14.64% 1.40GByteInst	40KByte 3.37% 548MByteInst	4KByte 2.65% 43.2MByteInst	124Byte 48.43% 23.1MByteInst

Die erste Spalte der Tabelle enthält die Information, welcher Compiler mit welchen Optionen für das entsprechende Experiment verwendet wurde. Die zweite gibt an, wieviele Instruktionen jeweils zur Abarbeitung des Dhystone-Benchmarks notwendig waren. Da die M88100-CPU im allgemeinen pro Takt eine Instruktion ausführen kann, ist dies gleichzeitig ein Maß für die Dauer der Berechnung.

Die nachfolgenden Tabellenelemente enthalten jeweils drei Zahlen:

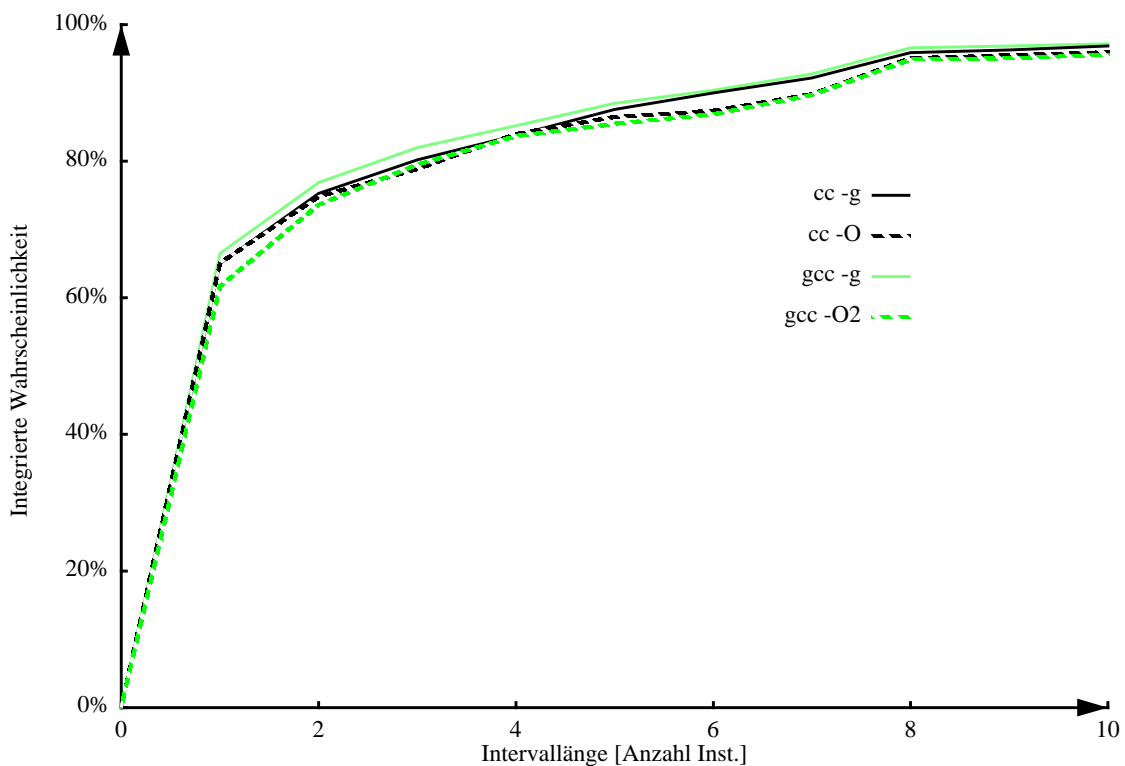
Die oberste Zahl gibt an, wieviele Bytes Speicherplatz das Textsegment, Datensegment und das Stacksegment des Prozesses beinhalten (jeweils Vielfache der Seitengröße 4 KByte) und wieviele Bytes Speicher die Registerbank enthält. Es ist zu sehen, daß je nach Compiler und Compiler-Option zum Teil unterschiedlich viel Hauptspeicher für die einzelnen Segmente des Prozesses reserviert werden. Die zweite Zahl ist das Maß, wieviel Prozent der darüber angegebenen Bytes im zeitlichen Mittel jeweils mit später noch benötigten Daten belegt war. Als drittes ist das Produkt aus der mittleren Anzahl der genutzten Speicherzellen und der Anzahl der notwendigen Schritte zur Beendigung des Benchmarks angegeben (Maßeinheit: Anzahl Bytes \* An-

### 3. Effiziente Experimentdurchführung

zahl Instruktionen). Dies ist ein wertvolles Maß zur Beurteilung der Empfindlichkeit eines Systems gegenüber Bit-Flip-Fehlern, da die Empfindlichkeit proportional zur Menge der gespeicherten, später wiederverwendeten Daten und proportional zur Dauer dieser Speicherung ist.

Der verwendete Compiler und sogar die verschiedenen Compiler-Optionen haben einen entscheidenden Einfluß auf die gemessenen Werte für den Nutzungsgrad der einzelnen Speicherelemente. So ist zum Beispiel im Mittel das Produkt aus genutztem Speicher und der Laufzeit für alle gemessenen Speicherelemente beim durch den GNU-Compiler ohne Optionen übersetzten Dhrystone-Benchmark etwa doppelt so groß wie im Falle einer Übersetzung mit Optimierungsoption.

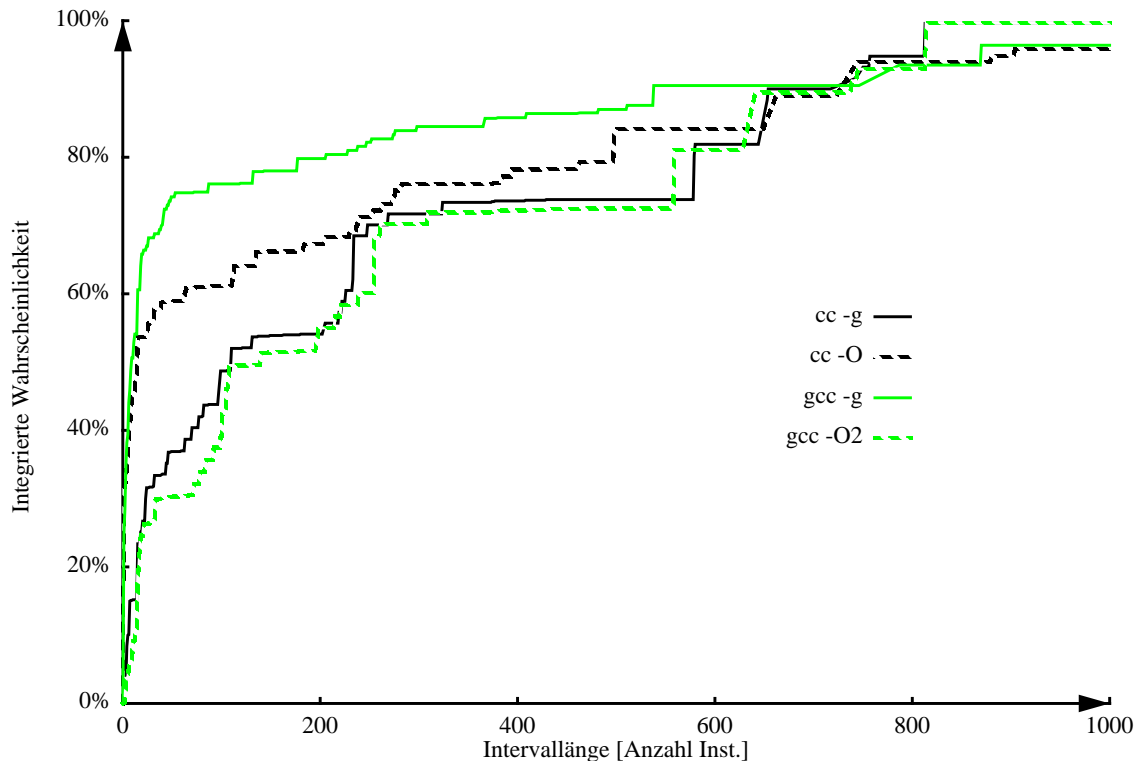
Das Diagramm in Abbildung 34 zeigt die Verteilung der zeitlichen Längen der mit einem Lesezugriff abgeschlossenen Intervalle für die General-Purpose-Register.



**Abb. 34: Compiler-Abhängigkeit der Länge der Zugriffsintervalle (Register-File)**

Es ist zu sehen, daß weit über 90% aller Intervalle, in denen die einzelnen General-Purpose-Register Daten enthalten, die später wiederverwendet werden, kürzer als 10 Taktzyklen sind. In ca. 65% aller Fälle werden die Daten sogar im nächsten Taktzyklus wieder gelesen. Die einzelnen Graphen für die unterschiedlichen Compiler und Compiler-Optionen unterscheiden sich kaum.

Im Vergleich dazu zeigt Abbildung 35 die Verteilung der Intervalllängen bei Zugriffen auf das Stack-Segment des Prozesses.



**Abb. 35: Compiler-Abhängigkeit der Länge der Zugriffsintervalle (Stack-Segment)**

Es ist zu sehen, daß die Intervalllängen im Mittel deutlich größer sind als die Längen bei Zugriffen auf das Register-File. Im Falle einer RISC-CPU mit einem großen Register-File ist dies auch zu erwarten. Es zeigt sich jedoch, daß die Längen je nach benutztem Compiler deutlich unterschiedlich sind. Da im Programm mehrere Unterprogrammaufrufe vorkommen, zu deren Ausführungsbeginn die CPU eine Reihe von Registerinhalten auf den Stack schreibt, die sie nach Ende der Routine zurückliert, und da die einzelnen Routinen mehrfach durchlaufen werden (hier 500 mal), enthalten die Graphen Sprünge.

Im zweiten Beispiel wurden verschiedene Programme untersucht: ein Dhrystone-Benchmark (*dhry*), ein Programm zur Lösung der zweidimensionalen Poisson-Differentialgleichung auf Basis eines Mehrgitterverfahrens (*mgall*; Beschreibung siehe z.B. [Hönig94]) und ein Programm mit einer einfachen Zählschleife *busy* (siehe Abbildung 36). Das Programm zur Lösung der Poisson-Differentialgleichung wurde weiterhin noch mit selbstüberwachendem Code (siehe [Hönig94]) versehen (*mgall.wpp*).

```

for (i = 0; i < count; i++) {
    /* do nothing */
}

```

```

                                or    r9,r0,0
                                br    @L22
                                addu  r9,r9,1
                                @L22:
                                cmp   r10,r23,r9
                                bbl   gt,r10,@L21

```

**Abb. 36: Hauptschleife des busy-Programms (C-Code und M88100-Assembler)**

### 3. Effiziente Experimentdurchführung

Beim Dhrystone-Benchmark (dhry) und beim Programm busy wurde die Anzahl der Durchläufe durch die Hauptschleife variiert, bei den Mehrgitterverfahren (mgall und mgall.wpp) die Gittergröße. Alle Programme wurden mit dem GNU-C-Compiler mit Optimierungsoption übersetzt. Tabelle 4 zeigt die Ergebnisse:

**Tabelle 4: Mittlerer Nutzungsgrad von Speicherplätzen (Benchmark-Abhängigkeit)**

Programm & Parameter	Memory			Register
	Text	Data	Stack	
busy 0	1.04%	3.46%	0.48%	8.88%
busy 10	1.10%	3.49%	0.48%	8.95%
busy 100	1.15%	3.60%	0.48%	8.99%
busy 1000	1.46%	4.30%	0.47%	8.74%
busy 10000	2.09%	5.76%	0.41%	8.07%
dhry 0	3.47%	2.09%	0.98%	14.51%
dhry 10	9.28%	2.68%	1.79%	30.04%
dhry 100	13.78%	3.26%	2.51%	45.44%
dhry 500	14.64%	3.37%	2.65%	48.43%
mgall 3	14.81%	8.01%	1.13%	43.73%
mgall 4	18.11%	12.17%	1.12%	60.32%
mgall 5	19.38%	26.47%	1.22%	66.14%
mgall 9	- <sup>a</sup>	-	-	62.0%
mgall.wpp 3	22.12%	6.79%	1.80%	33.79%
mgall.wpp 4	25.27%	10.29%	1.99%	39.08%
mgall.wpp 5	26.20%	22.28%	2.14%	41.20%
mgall.wpp 9	-	-	-	48.0%

a. nicht gemessene Werte (Zeitaufwand)

Wie aus den obigen Tabellen ersichtlich ist, hängt der Nutzungsgrad einzelner Speicherelemente sehr vom laufenden Programm und seinen Parametern ab. Dies begründet die zum Teil großen Unterschiede bei den Ergebnissen von Fehlerinjektionsexperimenten mit verschiedener Last ([Iyer86], [Güthoff95]). So können beispielsweise maximal 8.07% der in Register injizierten Bit-Flip-Fehler den Ablauf des busy-Programms (Parameter 10000) stören, während die restlichen 91.93% der Fehler mit Sicherheit keine Auswirkungen zeigen werden. Dagegen liegt der Nutzungsanteil beim Dhrystone-Benchmark mit Parameter 500 bei 48.43%. Nur 51.57% der Fehler führen sicher nicht zu einem Ausfall. Nach diesen Zahlen ist daher anzunehmen, daß der Dhrystone-Benchmark (Parameter 500) wesentlich empfindlicher bezüglich Register-Fehlern reagiert als das busy-Programm (Parameter 10000).

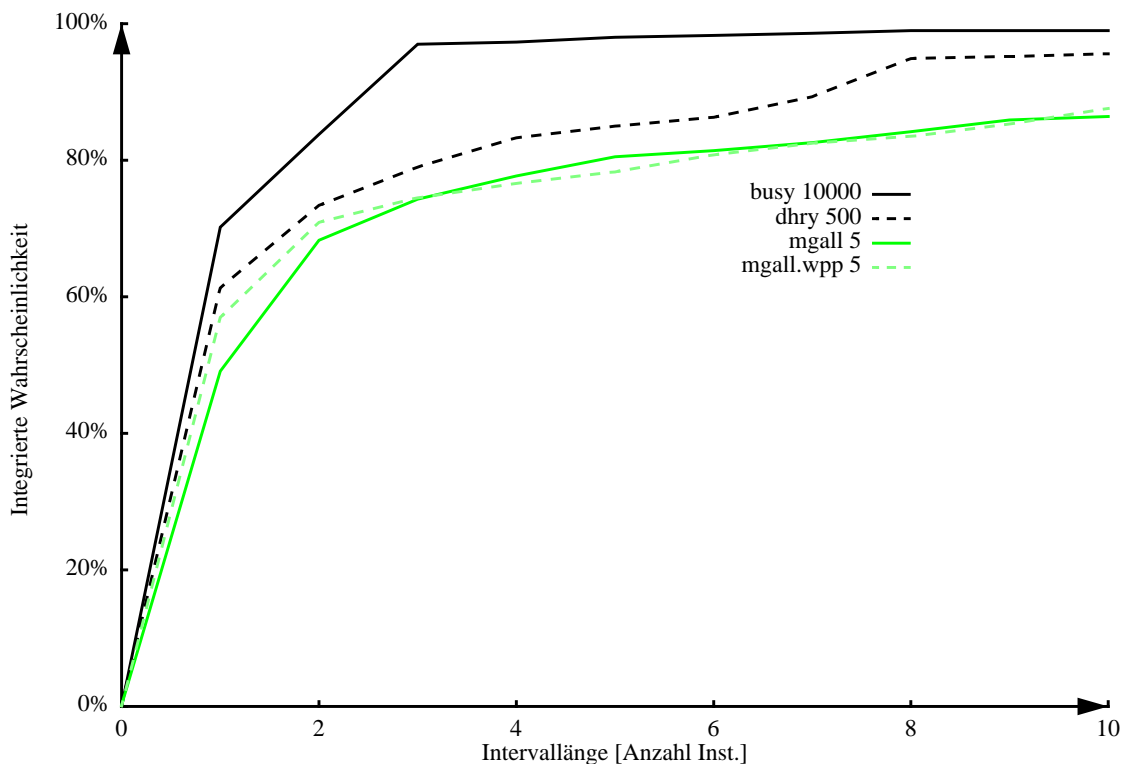
Wie die Tabelle zeigt, ist bei den meisten Anwendungen der Nutzungsgrad der einzelnen Speicherelemente im Mittel deutlich kleiner als 50%. In mehreren Fällen (z.B. auch in den nicht-künstlichen Anwendungen `mgall` bzw. `mgall.wpp`) liegt der Nutzungsgrad einiger Segmente (Stack-Segment) sogar deutlich unter 10%.

Aus der obigen Tabelle ist z.B. zu entnehmen, daß nur maximal 62% aller Registerfehler während des Laufes des `mgall`-Programmes mit dem Parameter 9 zu einem falschen Ergebnis führen können (`mgall.wpp`: 48.0%). Um diese Maximalwerte mit den wirklichen Zahlen vergleichen zu können, wurden Fehlerinjektionsexperimente während des Laufes der Programme `mgall` und `mgall.wpp` (Parameter 9) vorgenommen. Tabelle 5 zeigt, daß die Maximalwerte tatsächlich grob mit der Empfindlichkeit dieses Systems gegenüber Fehlern in Speicherelementen übereinstimmen.

**Tabelle 5: Vergleich von Nutzungsgrad und Fehlerauswirkungswahrscheinlichkeit**

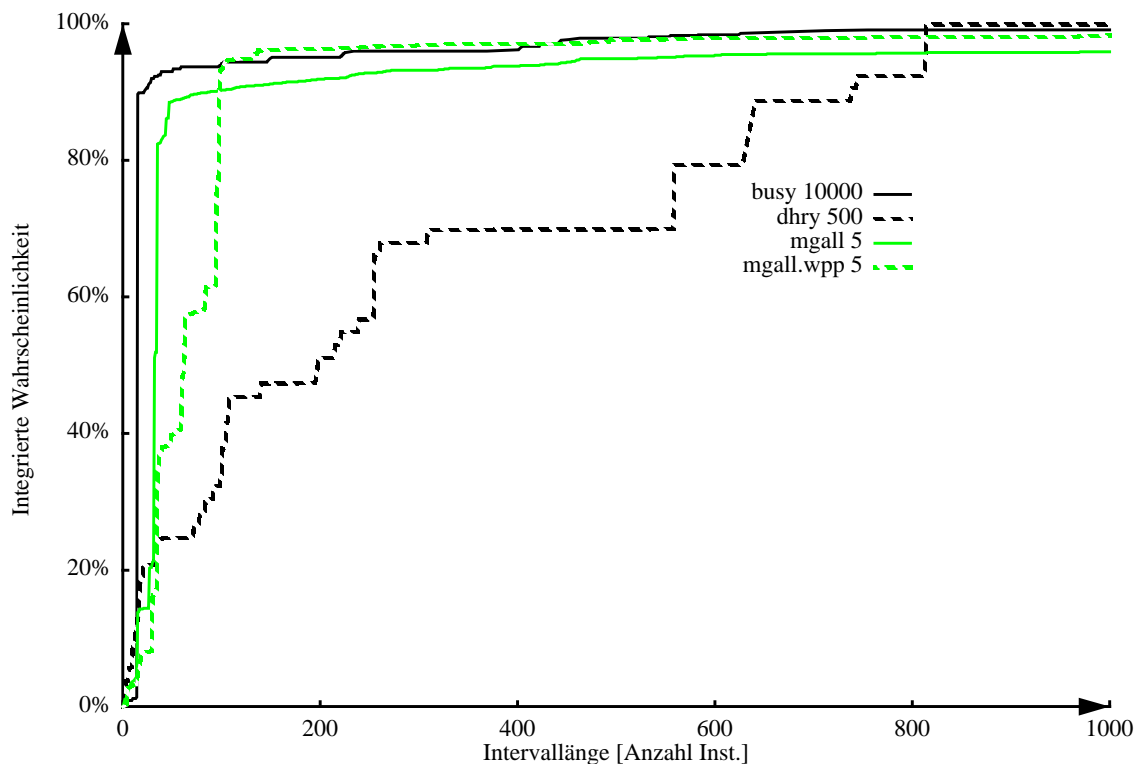
	mgall 9	mgall.wpp 9
Registernutzungsgrad	62.0%	48.0%
Ausfallwahrscheinlichkeit	56.0%	37.4%

Die nächsten Abbildungen zeigen die gemessenen Verteilungen der einzelnen Lebensdauern der Daten in den General-Purpose-Registern (Abbildung 37) bzw. der Daten im Stack-Segment (Abbildung 38).



**Abb. 37: Benchmark-Abhängigkeit der Länge der Zugriffsintervalle (Register-File)**

### 3. Effiziente Experimentdurchführung



**Abb. 38: Benchmark-Abhängigkeit der Länge der Zugriffsintervalle (Stack-Segment)**

Es ist aus den Diagrammen zu ersehen, daß die Verteilung der Intervalllängen stark von der Art der Benchmark-Programme abhängt. So hält das `busy`-Programm praktisch nur Daten in den Registern, die nach spätestens drei Taktzyklen wieder verwendet werden (Schleifenlänge: 3 Anweisungen). Die anderen getesteten Programme halten dagegen über längere Zeiträume Daten in den einzelnen Registern (flacherer Anstieg der Graphen für `dhry`, `mgall` und `mgall.wpp`).

Abbildung 38 zeigt, daß die Programme `busy`, `mgall` und `mgall.wpp` hauptsächlich Daten über kürzere Zeiträume auf dem Stack zwischenspeichern. Der Dhrystone-Benchmark (`dhry`) verwendet Daten auf dem Stack dagegen über längere Zeiträume. Die untere Schranke für die mittlere Fehlerlatenzzeit für Bit-Flip-Fehler im Stack-Segment ist daher beim Dhrystone-Benchmark größer als bei den anderen getesteten Programmen.

#### 3.8.4 Bewertung des Verfahrens

Um den Nutzungsgrad oder die Nutzungszeiträume einzelner Speicherzellen zu bestimmen, muß das System im Einzelschrittmodus betrieben oder simuliert werden (siehe Abschnitt 3.8.2).

Der Einzelschrittmodus erfordert einen hohen Rechenaufwand im Vergleich zum normalen Programmablauf. Der Aufwand entsteht, da nach jedem einzelnen Schritt der Zustand der CPU gerettet und nach Bearbeitung der Ausnahmebehandlungsroutine wiederhergestellt werden muß. Während der Ausnahmebehandlung zwangsläufig modifizierte Register (z.B. PC) sollen nicht den Ablauf des Testprogrammes verändern. Dieser Aufwand ist je nach CPU-Typ unterschiedlich, wird sich jedoch etwa im Bereich eines Faktors von 100 [Tschäche96a] bewegen. Wird die UNIX-`ptrace`-Schnittstelle<sup>1</sup> für diese Aufgabe verwendet, ist der Overhead - bedingt durch den zusätzlichen, doppelten Prozeßwechsel - noch größer. Die Analyse, welche Spei-

cherzellen bei den einzelnen Schritten gelesen bzw. geschrieben werden, ist dagegen relativ einfach. Sie erfordert im wesentlichen nur eine Dekodierung der nächsten Instruktion. Um mit Hilfe des Einzelschrittmodus eine Speicher- und Registernutzungsanalyse durchzuführen, benötigt man daher – je nach System – etwa zwischen 100 und 200-mal mehr Rechenzeit. Im obigen Beispiel des MEMSY-Systems dauerte eine Berechnung mit Analyse im Durchschnitt 189-mal länger als ein normaler Programmablauf.

Wird das zu untersuchende System simuliert, tritt keinerlei erhöhter Simulationsaufwand auf, da alle benötigten Daten während der Simulation des Golden-Runs ermittelt werden. Der Golden-Run ist aber für Funktionstests und ähnliches sowieso erforderlich. Es reicht aus, die Spuren der Taktsignale bzw. Output-Enable-Signale der entsprechenden Speicherzellen aus der Trace-Datei zu lesen. Diese Arbeit ist auch bei größeren Modellen (z.B. DP32 in Kapitel 6) innerhalb weniger Sekunden durchführbar.

Wenn man eine Speicherelement-Nutzungsanalyse durchführt, wäre es auch möglich, während der gleichen Zeit ca. 100 bis 200 – mit Hilfe des in Abschnitt 3.4 („Multi-Threaded Fault-Injection“) vorgestellten Verfahrens sogar noch mehr – Fehlerinjektionsexperimente durchzuführen. Soll eine grobe Abschätzung der Empfindlichkeit eines Systems gewonnen werden, stellt sich daher die Frage, ob dieses Verfahren sinnvoll angewendet werden kann. Sind jedoch einmal alle Schreib- und Lesezeitpunkte ermittelt worden, können nach den obigen Meßwerten mehr als 50% aller weiteren Fehlerinjektionsexperimente eingespart werden, da ihr Ergebnis aufgrund der Kenntnis der Lebenszyklen der gespeicherten Daten vorausbestimmt werden kann. Dies Verfahren wird daher um so attraktiver, je mehr Bit-Flip-Fehler injiziert werden sollen.

Sind  $N$  Fehlerinjektionsexperimente durchzuführen, beträgt der Aufwand (unter Einsatz des Verfahrens nach Abschnitt 3.4 mit bis zum Ende des Benutzerprogrammes (Dauer  $T$ ) dauerndem Beobachtungsintervall) ca.  $NT/2$ . Wird zunächst eine Nutzungsanalyse durchgeführt, beträgt der Aufwand dafür  $100T$  bis  $200T$  Zeiteinheiten. Zusätzlich werden für die  $N$  durchzuführenden Fehlerinjektionsexperimente etwa  $NTU/2$  Zeiteinheiten benötigt, wenn man einen Speicherelement-Nutzungsgrad von  $U$  annimmt. Der Gesamtaufwand beträgt in diesem Fall daher  $100T + NTU/2$  bis  $200T + NTU/2$  Zeiteinheiten. Ist  $N$  sehr groß, läßt sich der Aufwand zu  $NTU/2$  Zeiteinheiten abschätzen. Die Zeit, die benötigt wird, um eine große Anzahl von Fehlerinjektionsexperimenten auszuführen, kann daher in etwa um den Faktor  $U$  gesenkt werden. In den obigen Beispielen war  $U$  meist deutlich kleiner als 50%. Es ist daher mit einer Beschleunigung um einen Faktor größer als 2 zu rechnen.

Ähnliches gilt für die Analyse eines Systems durch eine Simulation. Auch in diesem Fall wird sich eine Beschleunigung der Experimente um den Faktor  $U$  ergeben. Da aber kein Zusatzaufwand benötigt wird, um die Schreib- und Lesezugriffszeiten zu ermitteln, gilt dies auch schon für wenige durchzuführende Fehlerinjektionsexperimente.

## 3.9 Irrelevante Fehler in kombinatorischen Schaltungen

### 3.9.1 Prinzipielles Verfahren

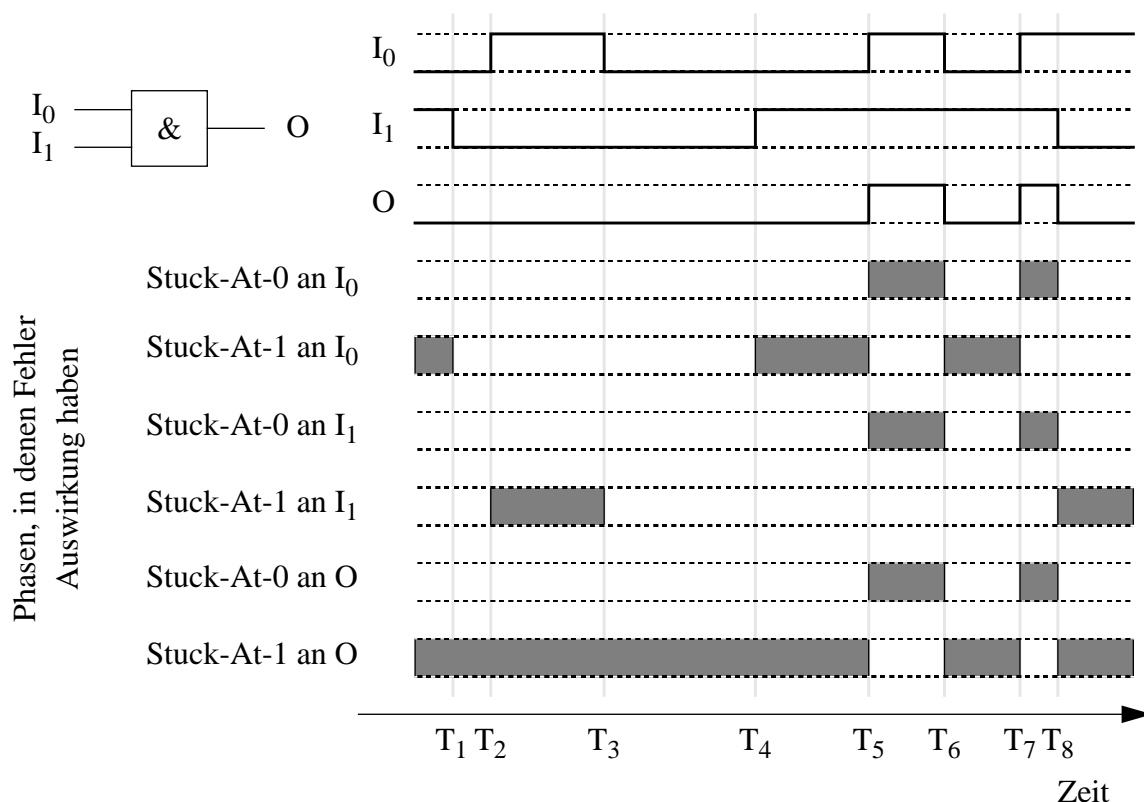
Bei Fehlerinjektionsexperimenten werden im allgemeinen Fehler in einzelne Komponenten injiziert. In vielen dieser Experimente wird der Fehler jedoch nicht einmal die Komponente verlassen, da injizierte Fehler häufig durch Redundanzen bzw. Maskierungen innerhalb der Komponente selbst behoben werden. Beispielsweise werden Fehler an einem Eingang eines AND-Gatters maskiert, wenn der andere Eingang während des Fehlerinjektionsintervalls mit einer lo-

---

1. Siehe UNIX-Manual „ptrace(2)“.

### 3. Effiziente Experimentdurchführung

gischen '0' belegt ist. Der Fehler wird daher nicht nach außen weitergegeben. Er ist außerhalb der Komponente nicht sichtbar. Dementsprechend wird das Gesamtsystem keinen Fehler bei seinen Berechnungen zeigen. Abbildung 39 verdeutlicht dieses Beispiel.



**Abb. 39: Phasen, in denen Fehler Auswirkungen haben**

Es ist zu sehen, daß beispielsweise zwischen den Zeitpunkten  $T_3$  und  $T_8$  ein Stuck-At-1-Fehler am Eingang  $I_1$  nach außen keine Auswirkungen haben wird, da während dieser Zeit entweder die Leitung sowieso schon auf logisch „1“ liegt (zwischen  $T_4$  und  $T_8$ ) oder  $I_0$  logisch „0“ führt und damit den Wert von  $I_1$  maskiert (AND-Gatter). Ein Stuck-At-1-Fehler des Eingangssignals  $I_1$  in einem Zeitintervall zwischen  $T_3$  und  $T_8$  wird daher in keinem Fall eine Störung des Gesamtsystems hervorrufen können, da der Fehler auf das AND-Gatter beschränkt ist.

Über einen Stuck-At-1-Fehler am Eingang  $I_1$  zwischen  $T_2$  und  $T_3$  läßt sich dagegen keine Aussage treffen, da in diesem Fall der Fehler nicht mehr auf das einzelne Gatter beschränkt ist, sondern sich über den Ausgang  $O$  auf nachfolgende Gatter ausbreiten kann.

Die Berechnung, ob ein Fehler sich über ein einfaches Gatter hinaus ausbreitet, ist schnell ausgeführt. Sie beschränkt sich auf wenige boolesche Ausdrücke. Abbildung 40 zeigt, wie dies am Beispiel des AND-Gatters berechnet werden kann. Die Booleschen Signale mit den Endungen „...\_critical“ (Deklarationen in den Zeilen 14 bis 19) haben jeweils den Wert TRUE, wenn ein Fehler dieser Art zum entsprechenden Zeitpunkt eine Auswirkung über das einzelne Gatter hinaus hätte. Zum Beispiel ist `sa0_i0_critical` dann TRUE, wenn ein Stuck-At-0-Fehler am Eingang  $i_0$  Auswirkungen auf den Ausgang  $O$  hätte (wenn  $(\text{'0'} \text{AND} i_1) \neq (i_0 \text{AND} i_1)$  ist; Zeile 24).

### 3.9 Irrelevante Fehler in kombinatorischen Schaltungen

---

```
1:ENTITY and_gate IS
2: PORT(   i0: IN  bit;
3:        i1: IN  bit;
4:        o:  OUT bit);
5:END and_gate;

6:ARCHITECTURE behaviour_faulty OF and_gate IS
7: SIGNAL  sa0_i0: boolean INTERVAL 1 year DURATION 20.0 ns;
8: SIGNAL  sa1_i0: boolean INTERVAL 1 year DURATION 20.0 ns;
9: SIGNAL  sa0_i1: boolean INTERVAL 1 year DURATION 20.0 ns;
10: SIGNAL sa1_i1: boolean INTERVAL 1 year DURATION 20.0 ns;
11: SIGNAL sa0_o  : boolean INTERVAL 1 year DURATION 20.0 ns;
12: SIGNAL sa1_o  : boolean INTERVAL 1 year DURATION 20.0 ns;
13:
14: SIGNAL sa0_i0_critical: boolean;
15: SIGNAL sa1_i0_critical: boolean;
16: SIGNAL sa0_i1_critical: boolean;
17: SIGNAL sa1_i1_critical: boolean;
18: SIGNAL sa0_o_critical : boolean;
19: SIGNAL sa1_o_critical : boolean;
20:BEGIN
21: PROCESS(i0,sa0_i0,sa1_i0,i1,sa0_i1,sa1_i1,sa0_o,sa1_o)
22:     VARIABLE tmp_o: bit;
23: BEGIN
24:     sa0_i0_critical <= ('0' AND i1) /= (i0 AND i1);
25:     sa1_i0_critical <= ('1' AND i1) /= (i0 AND i1);
26:     sa0_i1_critical <= (i0 AND '0') /= (i0 AND i1);
27:     sa1_i1_critical <= (i0 AND '1') /= (i0 AND i1);
28:     sa0_o_critical  <= '0' /= (i0 AND i1);
29:     sa1_o_critical  <= '1' /= (i0 AND i1);
30:
31:     IF sa0_i0 THEN tmp_o := '0';
32:     ELSIF sa0_i1 THEN tmp_o := '0';
33:     ELSIF sa1_i0 THEN tmp_o := i1;
34:     ELSIF sa1_i1 THEN tmp_o := i0;
35:     ELSE tmp_o := i0 AND i1;
36:     ENDIF;
37:     IF sa0_o THEN o <= '0';
38:     ELSIF sa1_o THEN o <= '1';
39:     ELSE o <= tmp_o;
40:     ENDIF;
41: END PROCESS;
42:END behaviour_faulty;
```

**Abb. 40: Berechnung der kritischen Fehler**

### 3. Effiziente Experimentdurchführung

---

Ähnlich diesem Beispiel lassen sich für alle kombinatorischen Schaltungen aus einem Golden-Run Daten über die einzelnen Phasen, in denen Fehler Auswirkungen haben können, gewinnen. Sind diese Phasen einmal bekannt, müssen nur noch diejenigen Fehlerinjektionsexperimente ausgeführt werden, deren Fehlerintervalle derartige Phasen überspannen. Alle anderen Fehler-simulationen brauchen nicht zeitaufwendig berechnet zu werden.

#### 3.9.2 Bewertung des Verfahrens

Die Berechnung der Signalverläufe der einzelnen Komponenten wird durch den Golden-Run ermöglicht. Dies ist im allgemeinen kein Zusatzaufwand, da ein Golden-Run zum normalen Funktionstest auch ohne Zuverlässigkeitsanalyse auf jeden Fall benötigt wird. Die zusätzliche Berechnung der „...\_critical“-Werte muß nur für die geplanten Fehlerinjektionsintervalle durchgeführt werden und kann damit praktisch vernachlässigt werden. Es ist sogar möglich, ihre Werte für jeden Zeitpunkt während des Golden-Runs mitberechnen zu lassen. Der Overhead beträgt nur ca. 2%, wie Messungen am Beispiel des DP32 (siehe Kapitel 6) gezeigt haben. Der Grund dafür ist, daß eine Prozeßumschaltung zwischen den verschiedenen Prozessen der Komponenten zeitlich sehr viel aufwendiger ist als die Berechnung der einzelnen Prozesse selbst (siehe auch Abschnitt 3.7). Es lohnt sich daher meistens nicht, einen größeren Aufwand für die Reduktion des Zeit-Overheads zu treiben. Werden jedoch nach Abschnitt 3.7 mehrere Prozesse zu einem zusammengefaßt, kann dieser Overhead leicht Größenordnungen von 100% erreichen. Eine Kombination dieser beiden Verfahren schließt sich daher weitgehend aus.

Größer als der Zeit-Overhead ist dagegen der Hauptspeicher-Overhead, da alle „...\_critical“-Signale zusätzlich gespeichert werden müssen. Zu jedem Fehlersignal muß genau ein „...\_critical“-Signal im Speicher gehalten werden. Im Fallbeispiel (Kapitel 6) beträgt der Overhead 84% (50058 Signale im Vergleich zu 27266 Signalen).

In der Trace-Datei des Golden-Runs werden nur Signaländerungen eingetragen. Wie am obigen Beispiel zu sehen ist, scheinen sich die „...\_critical“-Signale in etwa so häufig wie die Ein- und Ausgangssignale zu ändern. Im Falle von Stuck-At-Fehlern existieren jedoch weit mehr „...\_critical“- als normale Signale (jedes normale Signal ist mit mehreren Ein- und Ausgängen verbunden, während jeder einzelne Ein- oder Ausgang mehrere Fehler- und „...\_critical“-Signale besitzt). Es ist daher mit einem deutlichen Speicher-Overhead zu rechnen. Messungen am Beispiel des DP32 (siehe Kapitel 6) bestätigen dies. Die Trace-Datei vergrößert sich durch die Einführung der „...\_critical“-Signale um einen Faktor von ca. 4.2 (47.8 MBytes im Vergleich zu 9.19 MBytes).

Der Nutzen dieses Verfahrens ist schwieriger abzuschätzen. Hierfür muß zwischen sehr kurzen und längeren Fehlerinjektionsdauern unterschieden werden. Wenn die Dauer eines Fehlers deutlich kürzer ist als die Taktperiode eines synchron schaltenden Systems, kann man annehmen, daß der Fehler nur den Signalwert einer Periode stört. Für längere Fehlerdauern ist zu vermuten, daß sie zumindest teilweise auch in kritische Phasen fallen, in denen sie prinzipiell Auswirkungen auf übergeordnete Komponenten haben können. Das heißt, daß mit Hilfe dieses Verfahrens für sehr lang andauernde Fehler meist nur in Ausnahmefällen eine Aussage darüber getroffen werden kann, ob sie Auswirkungen auf das Verhalten des Gesamtsystems haben. Dieser Ansatz ist in diesem Falle weitgehend nutzlos. Bei kurzen Fehlerdauern (deutlich kürzer als eine Taktperiode) jedoch kann sehr häufig eine Auswirkung des Fehlers über die Komponente hinaus und damit eine Störung des Gesamtsystems ausgeschlossen werden. Dieser Fall soll im folgenden am Beispiel eines AND-Gatters mit den zwei Eingängen *A* und *B* und dem Ausgang *Q* mit Stuck-At-Fehlern genauer untersucht werden.

### 3.9 Irrelevante Fehler in kombinatorischen Schaltungen

Sei  $P_{XV}$  die Wahrscheinlichkeit, daß am Eingang  $X$  eines AND-Gatters mit zwei Eingabesignalen der Wert  $V$  anliegt,  $P_{EXV}$  die Wahrscheinlichkeit, daß der Eingang oder Ausgang  $X$  des AND-Gatters ein Stuck-At- $V$  Fehler besitzt, dann gilt für die Wahrscheinlichkeit  $P_{\text{no Effect}}$ , daß der Ausgang trotz eines Fehlers den richtigen Wert zeigt:

$$\begin{aligned} P_{\text{no Effect}} = & P_{A0}P_{B0} (P_{EA0} + P_{EA1} + P_{EB0} + P_{EB1} + P_{EQ0}) \\ & + P_{A0}P_{B1} (P_{EA0} + P_{EB0} + P_{EB1} + P_{EQ0}) \\ & + P_{A1}P_{B0} (P_{EA0} + P_{EA1} + P_{EB0} + P_{EQ0}) \\ & + P_{A1}P_{B1} (P_{EA1} + P_{EB1} + P_{EQ1}) \end{aligned}$$

Besitzen die sechs verschiedenen Fehlermöglichkeiten die gleiche Wahrscheinlichkeit, ist  $P_{\text{no Effect}}$  immer größer als 0.5. Im Falle zusätzlich gleichverteilter Eingangswerte gilt:  $P_{\text{no Effect}} \approx 0.66$ . Das heißt, daß man davon ausgehen kann, daß weniger als 50% der injizierten, kurzen Fehler eine Auswirkung haben werden. Messungen am Modell des DP32 (siehe Kapitel 6) bestätigen dies. Beim DP32 beträgt der Wert der Wahrscheinlichkeit 42.3%, der Faktor der Beschleunigung der Injektionsexperimente mit kurzzeitigen Fehlern daher 2.36.

Ähnliches gilt für andere kombinatorische Blöcke. Die Wahrscheinlichkeit, daß ein nur kurz andauernder Fehler eine Änderung des Ausgangssignals erzeugt, wird im Mittel etwas kleiner als 0.5 sein. Der Beschleunigungsfaktor der Fehlerinjektionsexperimente mit Fehlern kurzer Dauer mit Hilfe dieses Verfahrens wird daher im allgemeinen etwas über zwei liegen. Für länger dauernde Fehler wird jedoch keine wesentliche Verbesserung der Leistung zu erwarten sein.

Im Vergleich mit dem in Abschnitt 3.6 („Vergleich mit Golden-Run“) beschriebenen Verfahren sollte das hier beschriebene daher nur in Ausnahmefällen Verwendung bei der Beschleunigung von Fehlerinjektionsexperimenten finden. Ein paralleler Vergleich des fehlerhaften und des fehlerfreien Systems gibt eine wesentlich bessere Leistung.

Dieses Verfahren ist jedoch auch geeignet, eine Art Sensitivitätsanalyse des aufgestellten Modells bezüglich Fehlern zu erstellen. Anhand der „...critical“-Signale kann lokal und ohne Fehlerinjektion festgestellt werden, welche Fehler prinzipiell häufig zu Ausfällen führen können und welche seltener. Damit ist eine untere Schranke der Fehlermaskierungswahrscheinlichkeit kurzer Fehler berechenbar. Dies kann unter Umständen für eine grobe Bewertung eines Systems nützlich sein.



## 4. Detaillierte Experimentauswertung

Während die Fehlersimulation durchgeführt wird, sollen die Auswirkungen der injizierten Fehler beobachtet und protokolliert werden. Da die Auswirkungen nicht im voraus bekannt sind, ist eine möglichst vollständige Beobachtung und Protokollierung wünschenswert, um nach dem Experiment möglichst viele Fragen des Benutzers an das Modell und seine Auswertung beantworten zu können. Sind bestimmte Daten nicht protokolliert, die zur Beantwortung von Fragen notwendig gewesen wären, müssen die fraglichen Experimente mit genaueren Beobachtungsmöglichkeiten wiederholt werden. Dies bedeutet einen Zeitverlust und ist auch nur im Falle deterministischer Systeme überhaupt möglich.

Sind alle Experimente durchgeführt, müssen in einem zweiten Schritt die aufgezeichneten Spuren gemäß den Fragen des Benutzers ausgewertet werden. Sehr spezielle Fragen sind nicht durch ein universelles Werkzeug zu beantworten. Daten über allgemein bedeutsame Werte wie zum Beispiel mittlere Ausfallrate des Systems, mittlere Recovery-Zeit, mittlere Fehlerlatenzzeit u.ä. sollten jedoch von einem guten Werkzeug auf jeden Fall bereitgestellt werden können.

Abbildung 41 stellt das Zustandsübergangsdiagramm eines Systems dar, das dauerhaft eine bestimmte Leistung erbringen soll. Als Beispiel sei hier eine Fahrstuhlsteuerung genannt, die zyklisch die Ruftasten der einzelnen Stockwerke abfragen und den Fahrstuhl-Motor sowie den Motor für den Türmechanismus entsprechend ansteuern soll. Nach einem Reset wird dieser Zyklus von Zustandsübergängen meist nach kurzer Zeit erreicht und danach nicht wieder verlassen. Es sind jedoch auch andere Zyklen denkbar, die vom Benutzer als erlaubt angesehen werden. So kommt es beispielsweise nicht auf die Reihenfolge an, in der eine Fahrstuhlsteuerung die Ruftasten abfragt. Daher sind i.a. mehrere Zyklen erlaubt. Diese sind im Diagramm (Abbildung 41) grau hinterlegt.

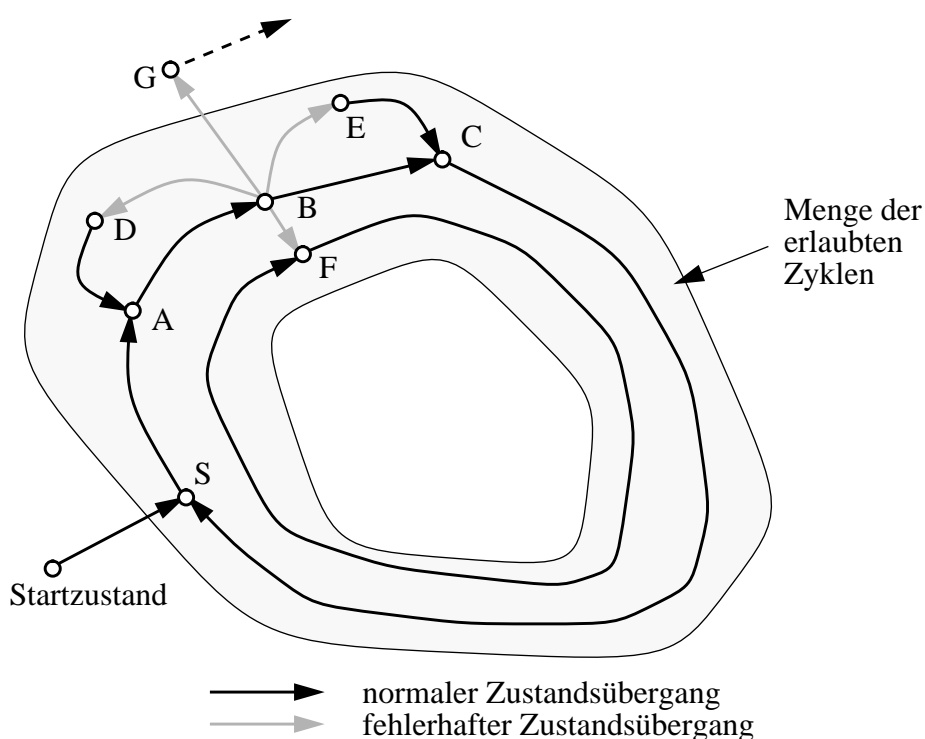
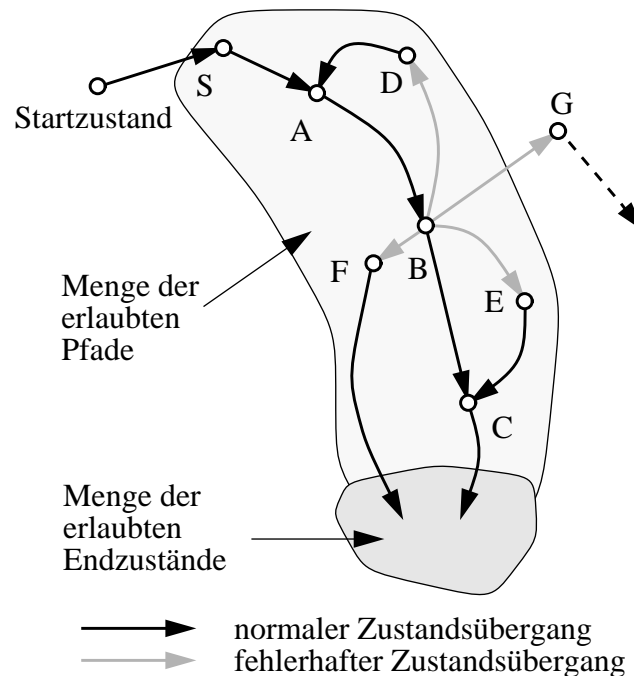


Abb. 41: Normale und fehlerhafte Zustandsübergänge in zyklisch arbeitenden Systemen

#### 4. Detaillierte Experimentalauswertung

Ein System, das ein einzelnes, zeitlich begrenztes Problem lösen soll (z.B. Wettervorhersage für den nächsten Tag, Lösung einer wissenschaftlichen Problemstellung), wird von Abbildung 42 gezeigt. Derartige Systeme erreichen nach einem Reset einen Pfad von Zustandsübergängen, der ohne Fehlereinwirkungen zum gewünschten Ergebnis und zu einem definierten Endzustand führt. Im allgemeinen führen viele verschiedene Pfade von Zustandsübergängen eines Computer-Systems zum selben Ergebnis. Diese sind aus Sicht des Systembenutzers alle erlaubt, da es ihm nicht auf den Weg der Berechnung, sondern auf das Endergebnis ankommt. Daher gibt es eine Menge von verschiedenen Berechnungspfaden (im Diagramm hellgrau unterlegt) und eine Menge von Endzuständen (im Diagramm dunkelgrau markiert), die erlaubt sind.



**Abb. 42: Normale und fehlerhafte Zustandsübergänge in einmalig arbeitenden Systemen**

Die Systeme durchlaufen diese Reihe von Zuständen zyklisch bzw. einmalig. Dieser Ablauf kann jedoch aufgrund von Fehlern beeinflusst werden. Je nach Art und Zeitpunkt eines Fehlers kann die Beeinflussung zum dauerhaften Ausfall, zeitweisen Ausfall oder auch zu keinerlei Ausfallserscheinungen führen.

In beiden Beispieldiagrammen durchläuft das System (in Abbildung 41 wiederholt) die Zustände A, B, C. Im Falle eines Fehlers zu einem Zeitpunkt, zu dem sich das System im Zustand B befindet, sind drei verschiedene Reaktionen des Systems denkbar:

- Das System geht in den Zustand D oder E über, von dem aus es zu einem nachfolgenden Zeitpunkt den erwünschten Zustand C trotz des Fehlers erreicht.
- Das System geht aufgrund des Fehlers in einen anderen Zyklus bzw. Pfad über (Zustand F), auf dem es jedoch auch das gewünschte Systemverhalten zeigt.
- Das System gerät in einen Zustand, der nicht wieder zur Menge der erwünschten Zyklen bzw. Pfade zurückführt (im Beispiel Zustand G).

Die Aufgabe der Auswertung ist nun, anhand der protokollierten Spuren zu bestimmen, ob ein System aufgrund eines Fehlers die Menge der erwünschten Zyklen verläßt bzw. nicht das gewünschte Endergebnis liefert. Wenn das System die Zyklen (Pfade) kurzzeitig verläßt, soll die

Zeitdauer bestimmt werden, die vergeht, bis das System wieder zu einem Zustand auf einem erwünschten Zyklus (Pfad) zurückkehrt. Bleibt das System trotz eines Fehlers innerhalb der Menge der Zustände innerhalb der gewünschten Zyklen bzw. Pfade, ist die zeitliche Versetzung des Programmablaufes zu berechnen.

In der Praxis ist es jedoch schwierig zu unterscheiden, ob ein durch ein Fehlerinjektionsexperiment berechneter Zustand auf dem erwünschten Pfad liegt. Um die Auswertung zudem ohne weiteren Programmieraufwand automatisch durchführen zu können, sind nur das gegebene Modell, die durch Fehlerinjektionen gewonnenen Spuren sowie die Spur des Golden-Runs zu verwenden. Im einfachsten Fall wird nur die Spur des Golden-Runs mit den einzelnen Spuren der fehlerhaften Läufe verglichen. Nur wenn sie identisch übereinstimmen, wird der injizierte Fehler als toleriert klassifiziert.

Abschnitt 4.1 beschreibt die in der Literatur bekannten Auswerteverfahren für Fehlerinjektionsexperimente. In den darauffolgenden Abschnitten 4.2 bis 4.5 wird auf die bestehenden Probleme genauer eingegangen und es werden Lösungen vorgestellt.

### 4.1 Bestehende Auswerteverfahren

Die Probleme bei der Auswertung von Fehlerinjektionsexperimenten werden in der Literatur häufig vernachlässigt. Bei den vorgestellten Experimenten wird in den meisten Fällen nicht auf die eigentliche Auswertung eingegangen. Häufig werden die Fehler nur klassifiziert (z.B. „no effect“, „wrong result“, „timeout“, „fail silent“, usw.). Wie jedoch genau von den Fehlersimulationsspuren auf diese Klassen geschlossen wurde, ist nicht beschrieben.

Die Verfahren, mit denen die Auswirkungen der Fehlerinjektionen protokolliert und ausgewertet werden, lassen sich grob in zwei Gruppen einteilen. Zur ersten Gruppe zählen alle Verfahren, die Experimente mit virtuellen, nicht real existierenden Systemen auswerten. Die zweite Gruppe wird von den Werkzeugen gebildet, die existierende Systeme instrumentieren und so an Daten über die Auswirkungen der Fehlerinjektionen gelangen. Die letzte Gruppe läßt sich weiter in Hardware- bzw. Software-basierte Beobachtungswerkzeuge unterteilen.

#### 4.1.1 Simulation virtueller Systeme

Wird ein virtuelles System auf einem Rechner simuliert, so existieren in dessen Arbeitsspeicher sämtliche Informationen über das System. Der gesamte Zustand des Systems liegt vor und kann zur späteren Verwendung zum Beispiel in eine Datei geschrieben werden. Dies ist ein sehr großer Vorteil der Simulation virtueller Systeme gegenüber den instrumentierten Systemen. Zu allen im Modell vorhandenen Zuständen (Variablen und Signale) sind die Daten auslesbar. Durch das Modell ist festgelegt, welche Daten prinzipiell verfügbar sind und welche nicht. Der Modellierer kann also frei bestimmen, welche Daten für ihn interessant sind.

Die Verfügbarkeit von einer Vielzahl verschiedener Daten kann jedoch auch zu einem Nachteil werden. Wird nicht ausgewählt, welche Daten mitprotokolliert werden sollen, sondern der gesamte Ablauf einer Simulation gespeichert, können – auch bei recht kleinen Systemen – schnell große Datenmengen anfallen.

Bei in der Literatur bekannten Fehlerinjektionen (z.B. [Jenn94]) werden die einzelnen Zustände, die das untersuchte System nach einer Fehlerinjektion durchläuft, mit denen verglichen, die während eines Golden-Runs aufgetreten sind. Es findet ein einfacher 1:1-Vergleich statt. Nur wenn alle Teilzustände mit dem Golden-Run übereinstimmen, gilt der Fehler als toleriert. Häufig auftretende, zeitliche Abweichungen (siehe Abschnitt 4.2) oder lokal auftretende, unwesent-

## 4. Detaillierte Experimentauswertung

---

liche Änderungen (siehe Abschnitt 4.3) führen dazu, daß ein Fehler als „nicht toleriert“ klassifiziert wird. Die gewonnenen Abschätzungen für die Fehlertoleranz sind daher nur recht grobe, obere Schranken.

### 4.1.2 Simulation instrumentierter Systeme

Werden bestehende Komponenten mit einer Instrumentierung versehen, können während eines Fehlerinjektionsexperiments an den instrumentierten Stellen im System die zeitlichen Veränderungen von Teilen des Systemzustandes mitprotokolliert werden. Aufgrund des großen Software- bzw. Hardware-Aufwandes, den eine derartige Instrumentierung kostet, wird im allgemeinen nur ein Teil des Systems instrumentiert. Die Systeme sind daher nicht vollständig beobachtbar. Deshalb können derartige Fehlerinjektionsexperimente auch nicht eindeutig ausgewertet werden. Es sind entsprechend nur obere bzw. untere Schranken für die gemessenen Werte anzugeben.

#### Hardware-instrumentierte Systeme

Wird die Instrumentierung mit Hilfe von Hardware vorgenommen, werden in den in der Literatur bekannten Experimenten (**FTMP** [Lala83], **MESSALINE** [Arlat90], **FOCUS** [Choi92], **HYBRID** [Young92b], **RIFLE** [Madeira94] und **SCRIBO** [Steininger97]) eine Reihe von Meßkontakten auf die Hauptplatine des Systems aufgesteckt. Die Anzahl der Kontakte ist jedoch beschränkt. Zum einen sind praktisch nur äußere Pins der einzelnen integrierten Bausteine sowie die Leiterbahnen zwischen den einzelnen IC's zugänglich. Zum anderen ist die Anzahl durch den zur Verfügung stehenden Platz zum Anstecken der Tastköpfe begrenzt. Dieses Problem wird immer größer mit der zunehmenden Miniaturisierung der einzelnen Komponenten. In der Praxis ist die Anzahl der Meßpins daher stark eingeschränkt.

Wie die so gewonnenen Spuren der Fehlerinjektionsexperimente weiterverarbeitet werden, ist in keiner bekannten Veröffentlichung näher beschrieben. Gespräche mit den betroffenen Wissenschaftlern zeigen, daß an dieser Stelle meist eine Auswertung per Hand durchgeführt wird. Die interessanten Spuren werden aufgrund von zusätzlich durchgeführten Experimenten mit Software-instrumentierten Systemen ausgesucht (z.B. **RIFLE** [Madeira94]).

#### Software-instrumentierte Systeme

In den meisten, durch Software instrumentierten Systemen werden die Zeitpunkte notiert, zu denen die CPU des unter Beobachtung stehenden Systems eine Ausnahmebehandlung anstößt. Da viele, in die CPU oder den Speicher injizierte Fehler in Zugriffsfehlern („bus error“, „segmentation fault“, o.ä.) enden (siehe zum Beispiel **EFA** [Echtle92], **ProFI** [Lovric95], **SFI** [Rosenberg93], **CSFI** [Carreira95a], **FINE** [Kao93], **FIAT** [Barton90], **DOCTOR** [Han93], **XCEPTION** [Carreira95b] sowie [Sieh93] und [Sieh94]), bekommt man dadurch häufig einen groben Eindruck von den Auswirkungen der Fehlerinjektion.

Diese Art der Beobachtung ist meist leicht in bestehende Systeme zu integrieren. Es reicht, in die Ausnahmebehandlungsroutinen der CPU Anweisungen einzufügen, die eine kurze Nachricht mit der Art der Ausnahmebehandlung und der aktuellen Zeit an den Benutzer schicken. Ein weiterer Vorteil ist, daß die aufgezeichneten Spuren nur einen geringen Datenumfang haben. Nur bei erkanntem Fehler werden einige wenige Bytes Information erzeugt. Diese lassen sich leicht speichern und weiterverarbeiten.

Problematisch an dieser Art der Beobachtung ist, daß vom Programmierer des Systems gewollte Ausnahmebehandlungen (z.B. während einer Speicher-Testphase auftretende „bus errors“) die Auswertung der Fehlerinjektionsexperimente stören können, da es dem Signal allein nicht anzusehen ist, ob es vom Programmierer der Applikation gewollt ist oder es aufgrund der Fehlerinjektion aufgetreten ist.

Zusätzlich zu den mitgeschriebenen Ausnahmebehandlungen wird bei den Software-instrumentierten Systemen häufig die Ausgabe des Systems auf Veränderungen gegenüber den Ausgaben eines Golden-Runs überprüft. Zum Teil lassen sich dadurch Auswirkungen eines injizierten Fehlers erkennen (Abschnitt 4.4).

Auch diese Art der Instrumentierung hat den Vorteil, daß die protokollierten Daten von relativ geringem Umfang sind, da die inneren Vorgänge in einem System in der Praxis meist sehr viel komplexer sind als seine Ausgaben. Werden alle Daten über eine gemeinsame Schnittstelle ausgegeben (z.B. `write`-Aufrufe), lassen sich leicht Anweisungen in das System integrieren, die den Inhalt und den Zeitpunkt der Ausgaben mitprotokollieren.

Es ist jedoch schwierig, Ausgaben mitzuschreiben, die nicht über spezielle Ausgabefunktionen abgewickelt werden. Ausgaben mittels Memory-Mapped-IO lassen sich z.B. nur mit sehr großem Programmier- und Laufzeitaufwand ermitteln (siehe Abschnitt 4.4).

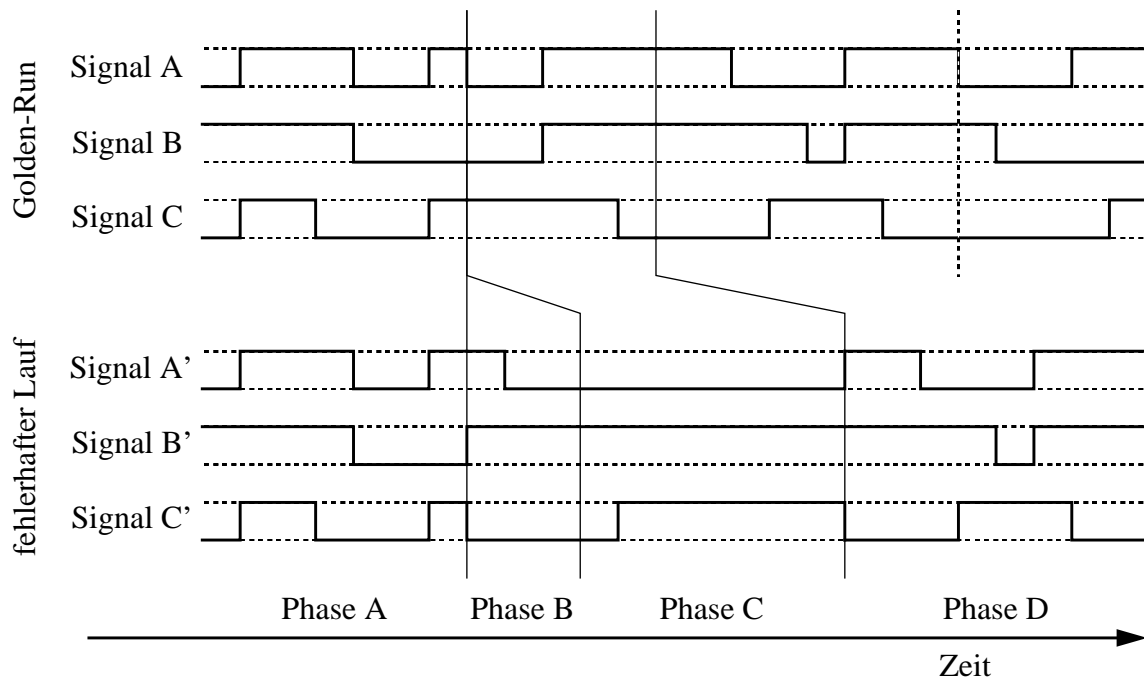
## 4.2 Zeitverschiebung durch Fehler

### 4.2.1 Darstellung des Problems

Wie viele Experimente gezeigt haben (z.B. auch Experimente mit dem DP32 in Kapitel 6), wird häufig das zeitliche Verhalten eines Systems durch Fehler beeinflusst. So kommt es häufig vor, daß bestimmte Zustände des Systems zwar wie im fehlerlosen Fall eingenommen werden, dies jedoch etwas früher oder später als im Normalfall geschieht (siehe auch Abbildung 41 und 42 die Zustände *D* und *E*). Einfache Beispiele für derartige zeitliche Verschiebungen sind kurzzeitige Fehler in der Takt/Reset-Erzeugung oder Fehler bei der Generierung von Signalen auf Handshake-Leitungen.

Da es in vielen Fällen jedoch nicht auf eine (kurze) zeitliche Verschiebung der Systemaktivitäten ankommt, sind derartige Reaktionen des Systems häufig als „zulässig“ zu klassifizieren. Andernfalls ist die durch die Experimente gewonnene Abschätzung der Fehlerauswirkungen eine gröbere, obere Schranke. Die Länge der erlaubten zeitlichen Verschiebung liegt – je nach untersuchtem Problem – bei unterschiedlichen Werten. Ein Kraftfahrzeug verträgt beispielsweise eine falsche Steuerfunktion eines ABS-Gerätes ca. eine Millisekunde lang. Die Fehlfunktion einer Steuerung eines Kernkraftwerkes kann ca. 20 Minuten anhalten. Erst danach ist ein Unglück nicht mehr aufzuhalten. Die tolerierbare Zeitspanne ist i.a. abhängig von der Trägheit der gesteuerten Anlagen.

Abbildung 43 zeigt an einem Beispiel eine derartige zeitliche Verschiebung der Systemaktivitäten.



**Abb. 43: Durch Fehler hervorgerufene Zeitverschiebung**

Während Phase A haben alle Signale des fehlerhaften Laufes den gleichen Wert wie im fehlerfreien Fall, da der Fehler bisher noch nicht aufgetreten ist. Mit Beginn der Phase B wird der Fehler aktiv und damit werden die Zustände der Systeme unterschiedlich. Im Falle eines temporären Fehlers verschwindet dieser nach einer gewissen Zeit (Ende von Phase B). Die Signale kehren jedoch nicht sofort wieder in ihren fehlerfreien Zustand zurück, da das Auftreten des Fehlers Auswirkungen hinterlassen hat. Diese sind erst mit Beginn der Phase D wieder aus dem System getilgt. Während der Phase C kann das Systemverhalten völlig unterschiedlich sein.

Da der Fehler selbst in das System injiziert wird und die Fehlerparameter (Zeitpunkt, Zeitdauer) bekannt sind, können die Längen der Phasen A und B ohne Untersuchung der aufgezeichneten Spuren angegeben werden. Die zeitliche Dauer der Phasen C im fehlerfreien und fehlerbehafteten Lauf ist jedoch nicht bekannt und sollte bestimmt werden.

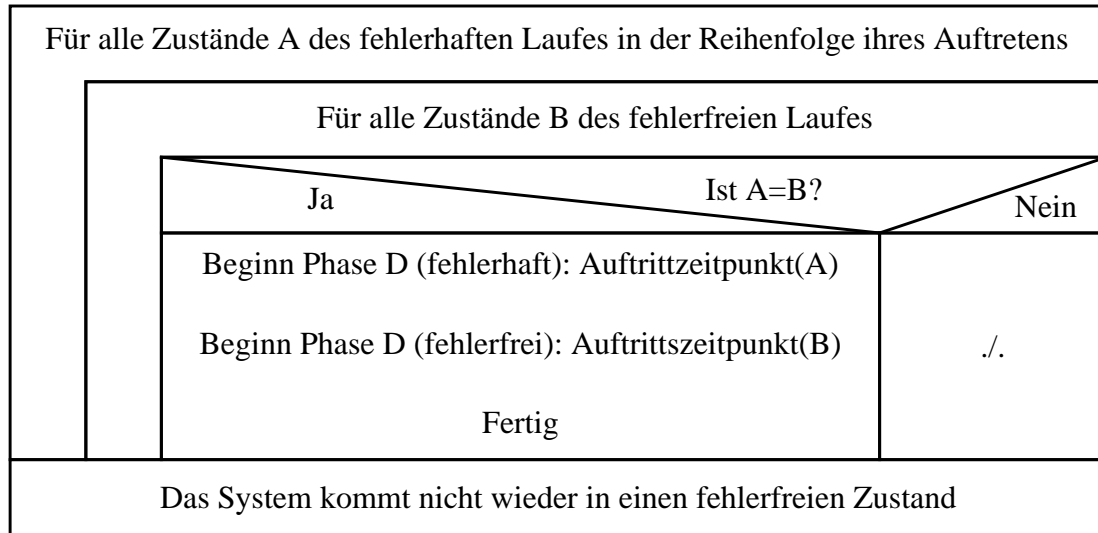
### 4.2.2 Algorithmus zur Bestimmung der Zeitverschiebung

Um die Länge der Phasen C zu bestimmen, ist es notwendig, den ersten Zustand nach der Fehlerinjektion zu bestimmen, der mit einem Zustand übereinstimmt, der während des Golden-Runs ebenfalls aufgetreten ist. Unter der Voraussetzung<sup>1</sup>, daß das Modell des untersuchten Systems deterministisch arbeitet, werden dann auch alle weiteren Zustände des fehlerhaften Laufes mit dem Golden-Run übereinstimmen.

Um den ersten wieder übereinstimmenden Zustand zu finden, sind zwei verschiedene Algorithmen denkbar.

<sup>1</sup> Ist diese Voraussetzung nicht erfüllt (z.B. bei Interrupts zu nicht durch eine Modellanalyse vorhersagbaren Zeitpunkten), ist ein Vergleich der Spuren ohne weiteres Wissen über das System ohnehin zum Scheitern verurteilt.

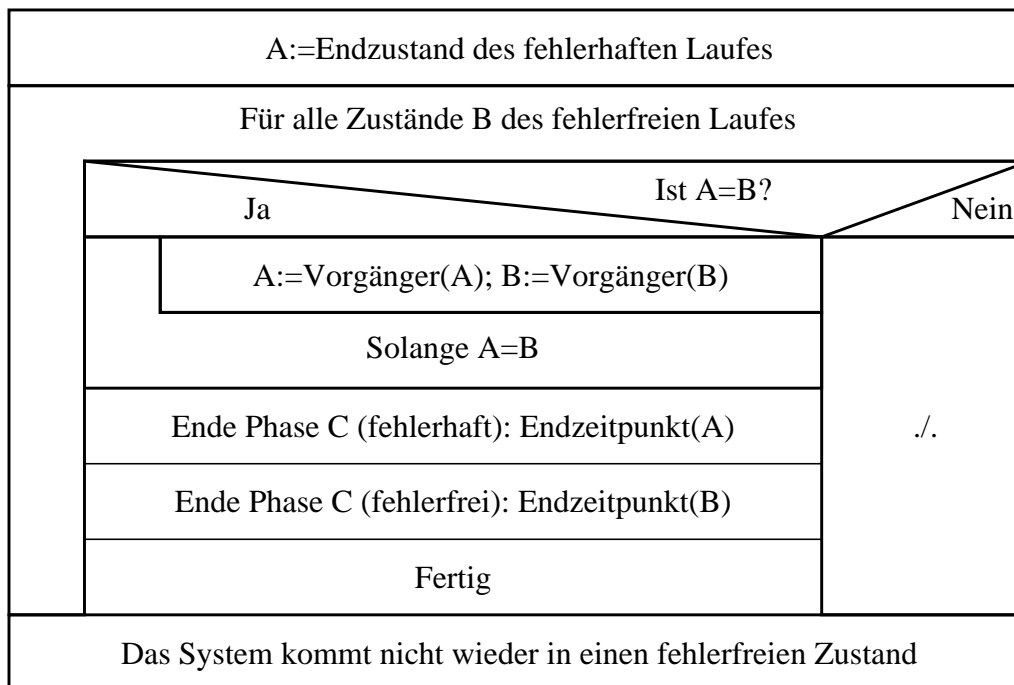
Der erste Lösungsansatz versucht, der Reihe nach alle Zustände des Testsystems nach dem Verschwinden des Fehlers mit Zuständen des Golden-Runs zu vergleichen. Sobald eine Übereinstimmung gefunden ist, bricht der Algorithmus ab. Die Auftrittszeitpunkte der zwei zuletzt miteinander verglichenen – und gleichen – Zustände repräsentieren den Beginn der Phasen *D*. Daraus kann die Zeitverschiebung berechnet werden. Abbildung 44 zeigt ein Struktogramm zur Beschreibung des Algorithmus.



**Abb. 44: Algorithmus A zur Berechnung der durch Fehler provozierten Zeitverschiebung**

Ein anderer Ansatz zur Berechnung der Zeitverschiebung versucht, nur den letzten in der Spur des fehlerhaften Laufes gespeicherten Zustand in der Spur des Golden-Runs wiederzufinden. Von dort ausgehend, können zeitlich rückwärts die einzelnen Zustände des Golden-Runs und des fehlerhaften Laufes verglichen werden, bis sie nicht mehr übereinstimmen. Die ersten nicht mehr gleichen Zustände kennzeichnen den Übergang von Phase *C* nach Phase *D*. Abbildung 45 zeigt das Struktogramm des Algorithmus.

#### 4. Detaillierte Experimentauswertung



**Abb. 45: Algorithmus B zur Berechnung der durch Fehler provozierten Zeitverschiebung**

#### 4.2.3 Bewertung

Besteht die Spur des Golden-Runs aus  $N$  und die Spur des damit zu vergleichenden, fehlerhaften Laufes aus  $M$  gespeicherten Zuständen (i.a. gilt:  $M \ll N$ , siehe Abschnitt 3.4), müssen nach dem ersten Algorithmus im Extremfall  $N \cdot M$  Zustandsvergleiche durchgeführt werden. Zwar werden im Falle tolerierter Fehler häufig deutlich weniger als  $N \cdot M$  Zustandsvergleiche benötigt, wird ein Fehler jedoch nicht vom System behoben, sind für diese Feststellung genau  $N \cdot M$  Vergleiche notwendig. Für selten tolerierter Fehler (z.B. Bit-Flip-Fehler im Fallbeispiel in Kapitel 6) werden damit im Mittel fast  $N \cdot M$  Vergleiche notwendig sein. Im besten Fall ist nur ein einziger Zustandsvergleich durchzuführen. Dieser Fall tritt jedoch nur ein, wenn der injizierte Fehler keinerlei Auswirkungen hatte.

Beim zweiten Algorithmus müssen zur Abarbeitung der ersten Schleife maximal  $N$  Zustandsvergleiche durchgeführt werden. Für die zweite Schleife werden maximal  $\min(N, M)$  Vergleiche benötigt. Da i.a. gilt, daß  $M \ll N$ , erhält man zur Abschätzung der maximalen Laufzeit des Algorithmus  $N + M$  Vergleichsoperationen. Hier gilt für den besten Fall, daß nur zwei Vergleiche benötigt werden: je einer für die innere bzw. äußere Schleife.

Im praktischen Einsatz wird die Anzahl der Vergleiche im Mittel zwischen dem Maximal- und Minimalwert liegen. Dies ist aber stark vom untersuchten System abhängig. Wie Versuche mit den beiden Algorithmen an verschiedenen Beispielen jedoch gezeigt haben, ist der zweite Ansatz dem ersten in der Praxis immer überlegen. Dies wird um so deutlicher, je mehr Fehler nicht toleriert werden, da dann die jeweilige maximale Anzahl von Vergleichen beim zweiten Verfahren etwa um den Faktor  $M$  niedriger liegt.

Die Genauigkeit von Ergebnissen von Fehlerinjektionsexperimenten kann mit Hilfe dieser Algorithmen etwas verbessert werden. Die genauen Werte sind natürlich abhängig von der Art des untersuchten Systems. Messungen am Beispiel des DP32 (siehe Kapitel 6) haben folgende Verbesserungen ergeben (Tabelle 6):

**Tabelle 6: Verbesserung der Genauigkeit durch Berechnung der Zeitverschiebung**

Hardware-Modell	Fehlermodell	ohne Einsatz des Verfahrens		mit Einsatz des Verfahrens	
		korrigierte Fehler	nicht korrigierte Fehler	korrigierte Fehler	nicht korrigierte Fehler
„Advanced“	temporäre, interne Stuck-At-Fehler	95.8%	4.22%	96.1%	3.94%
„Simple“	temporäre, interne Stuck-At-Fehler	95.6%	4.40%	93.9%	4.10%
„Advanced“	Bit-Flip-Fehler	22.6%	77.4%	24.2%	75.8%

Die ersten beiden Spalten beschreiben das verwendete Hardware- bzw. Fehlermodell (für eine Beschreibung der Modelle siehe Kapitel 6). Die letzten vier Spalten vergleichen die Abschätzungen der Ausfalls- und Überlebenswahrscheinlichkeiten der einzelnen Systeme ohne bzw. unter Einsatz des oben beschriebenen Verfahrens.

Wie an den Werten zu sehen ist, konnten die Abschätzungen bezüglich der Wahrscheinlichkeiten für einen Systemausfall bzw. für ein Systemüberleben zum Teil um knapp 10% verbessert werden.

### 4.3 Zustandsänderung durch Fehler

In vielen Fehlerinjektionsexperimenten ist zu erkennen, daß ein System – bedingt durch einen kurzzeitig injizierten Fehler – in einen fehlerhaften Zustand gerät. Häufig wird jedoch der fehlerhafte Teil des Zustandes nur für die Funktion von unwesentlichen Eigenschaften des Systems benötigt. In diesem Fall reicht es aus, das System als „defekt, aber nicht vollständig ausgefallen“ zu klassifizieren. In einigen Fällen ist es sogar erwünscht, daß das System nicht wieder in den einen Zustand zurückkehrt, in dem es im fehlerlosen Fall auch gewesen wäre. Man denke z.B. an Zähler, welche die Anzahl der bisher erkannten und behobenen Fehler registrieren. Zustandsvergleiche ähnlich denen im vorangehenden Abschnitt werden nicht erkennen können, daß sich das eigentliche System nach der Fehlerkorrektur wieder in einem korrekten Zustand befindet, da sich der Fehlerzähler vom fehlerfreien Fall unterscheidet.

Es ist daher für die Auswertung der Spuren notwendig, die Komponenten des Systems, die für den eigentlichen, gewünschten Betrieb der Anlage gebraucht werden, von den weniger wichtigen Teilen des Systems zu trennen.

### 4.4 Ausgabeänderung durch Fehler

An sich ist jeder Benutzer nur an den Ausgaben seines Computer-Systems interessiert. Ob der interne Zustand des Systems korrekt ist, ist für ihn nicht wichtig. Daher versuchen viele Fehlerinjektionswerkzeuge, anhand der Ausgabe eines Programmes festzustellen, ob ein injizierter

## 4. Detaillierte Experimentalauswertung

---

Fehler Auswirkungen hatte. Die Werkzeuge verändern daher das System so, daß alle Ausgaben mitprotokolliert werden. Die Ausgaben werden dann bei der Auswertung mit den Ausgaben des Golden-Runs verglichen.

Dieser Ansatz verspricht einen gewissen Erfolg im Falle von Anwendungen, die nur einmalig ein bestimmtes Ergebnis berechnen sollen (z.B. Rechenanwendungen). Da die Laufzeit von Systemen, die für Steuerungszwecke eingesetzt sind, jedoch nicht zeitlich begrenzt ist, sind auch die Ausgaben nicht in ihrer Datenmenge begrenzt. Ein Vergleich mit dem Golden-Run ist damit nicht möglich. Den Vergleich nach einer bestimmten Zeit abzurechnen, kann zu falschen Ergebnissen führen, da ein falscher interner Zustand des Systems sich unter Umständen erst nach sehr langer Zeit nach außen durch falsche Ausgaben bemerkbar macht. Der folgende Ansatz ist daher nur für Systeme geeignet, die einmalig bestimmte Ergebnisse produzieren sollen und danach einen Reset durchführen.

Durch Hardware instrumentierte Systeme können die Ausgaben dadurch mitprotokollieren, daß sämtliche Ausgabeleitungen an die Meßeingänge der Protokollierungseinheit angeschlossen werden. Im Falle einer Simulation eines virtuellen, nur als Beschreibung existierenden Systems kann die Aufzeichnung der Simulationsspuren einfach auf die Signale der Ausgabeleitungen begrenzt werden.

Aufwendig ist es jedoch, an die Ausgaben eines mit Software-Mitteln instrumentierten Systems heranzukommen. Zwar können die Ausgabefunktionen (z.B. `print`, `write`, `send`) so modifiziert werden, daß sie die auszugebenden Daten zusätzlich noch auf ein anderes Speichermedium schreiben, wo sie später mit den Spuren des Golden-Run verglichen werden können. Es ist jedoch schwierig, Ausgaben mitzuschreiben, die nicht über dedizierte Ausgabefunktionen abgewickelt werden. Ausgaben mittels Memory-Mapped-IO lassen sich dagegen nur mit sehr großem Programmier- und Laufzeitaufwand ermitteln.

Eine effiziente Möglichkeit, auch Ausgaben über Speicherzugriffe mitzuprotokollieren, besteht darin, die entsprechenden Speicherbereiche mit Hilfe der MMU vor unkontrollierten Schreibzugriffen zu schützen und damit unkontrollierte Ausgaben zu verhindern. Vor der eigentlichen Ausgabe kann dann die Protokollierung durchgeführt werden. Abbildung 46 zeigt Pseudo-C-Code zur Programmierung dieser Ausgabenprotokollierung.

```
void
init(...)
{
    ...
    /* programmiere MMU so, daß bei Schreibzugriffen */
    /* auf die IO-Bausteine Exception ausgelöst wird */

    mmu_write_protect(addr, ON);
    ...
    return;
}
```

```

void
write_protected_exception(...)
{
    ...
    /* hole Adresse und Wert */
    mmu_get_exception_address(&addr, &value);

    /* protokolliere Zugriff */
    log(addr, value);

    /* führe Schreibzugriff aus */
    mmu_write_protect(addr, OFF);
    *addr = value;
    mmu_write_protect(addr, ON);
    ...
    return;
}

```

**Abb. 46: Ausgabenprotokollierung mit Hilfe der MMU**

Diese Art der Ausgabenprotokollierung wurde ins MACH-Betriebssystem, [USENIX92], [USENIX93] implementiert [Probst94]. Es zeigt sich, daß eine Protokollierung der Ausgaben mit Hilfe dieses Ansatzes sehr effizient ist, wenn nur geringe Datenmengen zu übertragen sind. Dies ist einleuchtend, bedenkt man, daß nur dann, wenn wirklich Daten ausgegeben werden sollen, Ausnahmebehandlungen auftreten. Normale Rechenzyklen der CPU laufen mit unverminderter Geschwindigkeit ab.

## 4.5 Dynamische Anpassung der Zeitauflösung

### 4.5.1 Prinzipielles Verfahren

Um nach einer Fehlerinjektion zu entscheiden, wann der eingestreute Fehler aus dem System verschwunden ist, werden in gewissen Zeitabständen immer wieder Vergleiche mit dem ohne Fehler rechnenden System angestellt (siehe Abschnitt 3.6). Je häufiger diese Vergleiche durchgeführt werden, desto genauer kann die Fehlerlatenzzeit, Recovery-Zeit und ähnliches angegeben werden. Häufig ist jedoch keine absolute Genauigkeit (z.B. 1ns), sondern nur eine bestimmte, relative Genauigkeit (z.B. zwei Dezimalstellen) für die Zeitmessung gefordert. In diesem Fall kann durch eine geeignete Wahl der Vergleichszeitpunkte die Anzahl der Vergleiche deutlich gesenkt werden. Die Vergleiche sollten nicht in zeitlich äquidistanten, sondern in logarithmischen Abständen erfolgen.

*Definition „absolute Genauigkeit“:*

*Eine Messung besitzt eine absolute Genauigkeit  $R_a$ , wenn für einen Meßwert  $M$  eines realen Wertes  $W$  gilt:*

$$R_a \leq |M - W|.$$

#### 4. Detaillierte Experimentauswertung

Definition „relative Genauigkeit“:

Eine Messung besitzt eine relative Genauigkeit  $R_r$ , wenn für einen Meßwert  $M$  eines realen Wertes  $W$  gilt:

$$R_r \leq \left| \frac{M - W}{W} \right|.$$

Eine untere Schranke für die zeitliche Auflösung  $R_a = t_n - t_{n-1}$  ist durch das zu simulierende Modell oder die maximale zeitliche Auflösung des Simulators gegeben.

Folgende Zeitpunkte sind sinnvoll bezüglich der Anzahl der notwendigen Messungen bei gegebener, minimal möglicher, absoluter Genauigkeit  $R_a$  sowie gewünschter, relativer Genauigkeit  $R_r$ :

$$0, t_1, t_2, \dots, t_{n-1}, t_n, \dots$$

mit  $t_1 = R_a$  und  $t_n = (1 + R_r) t_{n-1}$ .

Durch Auflösung der Rekursion ergibt sich für  $t_n$ :

$$t_n = R_a (1 + R_r)^{n-1}.$$

Der Zeitpunkt 0 ist sinnvoll, um herauszufinden, ob ein injizierter Fehler überhaupt eine Auswirkung hatte oder vollständig maskiert wurde.

Für den nächsten Vergleich ist der Zeitpunkt  $t_1 = R_a$  am sinnvollsten, da ein früherer Zeitpunkt nicht möglich ist (bestmögliche zeitliche Auflösung des Simulators erreicht) und ein späterer Zeitpunkt die zeitliche Auflösung verschlechtert.

Alle nachfolgenden Vergleiche befinden sich dann im geometrischen Abstand  $1 + R_r$ , so daß die gewünschte, zeitliche, relative Genauigkeit gerade erfüllt wird.

Abbildung 47 zeigt am Beispiel  $R_a = 1ns$  und  $R_r = 0.01$  (zwei Dezimalstellen), zu welchen Zeitpunkten Vergleiche durchzuführen sind.

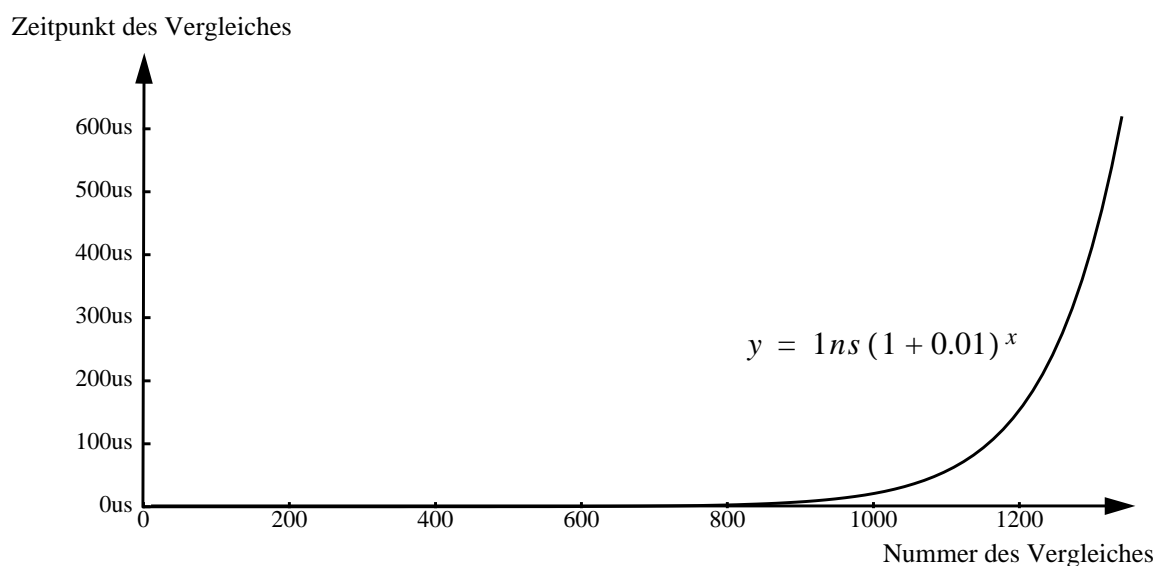
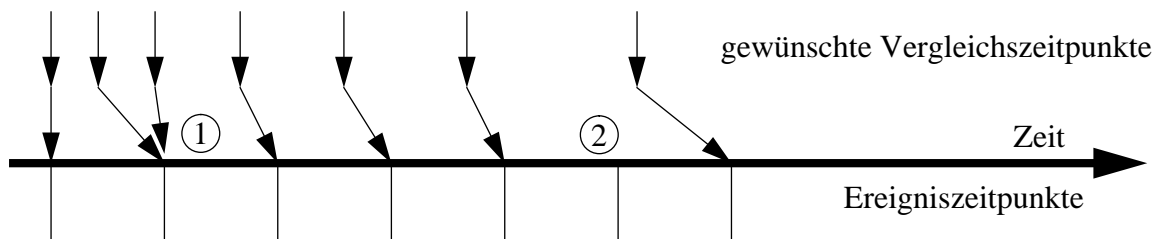


Abb. 47: Zeitpunkt eines Vergleiches in Abhängigkeit von seiner laufenden Nummer

In einer Ereignis-gesteuerten Simulation sind Zustände nicht jederzeit, sondern nur zu Zeitpunkten direkt beobachtbar, zu denen sich im System Teile des Zustandes ändern. Es ist jedoch stets bekannt, daß zwischen zwei Ereignissen keine Zustandsänderung auftritt. Daher kann jeweils der auf einen gewünschten Meßzeitpunkt folgende Zeitpunkt eines Ereignisses für den Zustandsvergleich verwendet werden. Unter Umständen fallen dadurch mehrere Messungen auf den gleichen Vergleichszeitpunkt. Dies kann besonders am Anfang der Messung (kurze zeitliche Abstände der Vergleichsmessungen) geschehen. Abbildung 48 zeigt am Beispiel, wie verschiedene Vergleichszeitpunkte auf unterschiedliche Ereigniszeitpunkte abgebildet werden. Dabei ist zum einen zu sehen, wie mehrere Vergleichszeitpunkte auf einen Ereigniszeitpunkt fallen können (1), und zum anderen, daß auf einige Ereigniszeitpunkte keine Meßzeitpunkte fallen (2).



**Abb. 48: Zuordnung der Meßzeitpunkte zu den Ereigniszeitpunkten**

In getakteten Systemen berechnet der kombinatorische Teil der Hardware zwischen den einzelnen Taktzyklen aus dem aktuellen Zustand des Systems den neuen Zustand. Diese Berechnung wird innerhalb eines Taktzykluses durchgeführt. Reicht für die zeitliche Auflösung der Messung eine Zeitauflösung, die größer ist als die Dauer eines Taktzykluses, müssen nicht mehr alle Signalverläufe überprüft werden. Dann reicht es aus, nur den neuen Zustand des Systems zu testen. Ist dieser korrekt, werden auch alle nachfolgenden Zustände korrekt berechnet werden, wenn kein Fehler im System ist.

Programmtechnisch ist es am einfachsten, jeweils zu Zeitpunkten Vergleiche durchzuführen, die ganzzahlige Vielfache der kleinsten Auflösung des Simulators sind ( $t_n = N_n R_a$  mit  $N_n \in \{0, 1, 2, \dots\}$ ).

Es sollte daher gelten  $t_n \leq (1 + R_r) t_{n-1}$  oder  $t_n = t_{n-1} + R_a$ . Da  $t_n$  möglichst groß gewählt werden sollte, um möglichst wenig Vergleiche durchführen zu müssen, sollte für  $t_n$  gelten:  $t_n = N_n R_a = \max(\lfloor (1 + R_r) N_{n-1} \rfloor, N_{n-1} + 1) R_a$ .

### 4.5.2 Bewertung des Verfahrens

Für den normalen Ablauf (ohne Anpassung) gilt:

Vergleiche werden zu den Zeitpunkten  $t_n = n R_a$  mit  $n \geq 0$  durchgeführt. Daraus ergibt sich für die Anzahl der Vergleiche  $N_n$  im Intervall  $[0, T]$ :

$$N_n = \left\lceil \frac{T}{R_a} + 1 \right\rceil.$$

#### 4. Detaillierte Experimentauswertung

---

Im Falle eines Ablauf mit dynamischer Anpassung der Genauigkeit gilt:

Die Vergleiche sollten zu den Zeitpunkten 0 und  $t_n = R_a (1 + R_r)^{n-1}$  mit  $n \geq 1$  durchgeführt werden. Somit ergibt sich für die Anzahl der Vergleiche  $N_d$  im Intervall  $[0, T]$ :

$$N_d \leq \left\lceil \frac{\ln \frac{T}{R_a}}{\ln(1 + R_r)} + 2 \right\rceil.$$

Dieser Wert ist nur eine obere Schranke. Dadurch, daß nicht jederzeit ein Vergleich durchgeführt werden kann, da die zur Zeit verwendeten Simulatoren eine beschränkte Zeitauflösung besitzen, sind weniger Vergleiche zu erwarten.

Für große Intervalle  $T$  und hohe Genauigkeiten  $R_a$  bzw.  $R_r$  gilt:

$$N_n \approx \frac{T}{R_a}$$

bzw.:

$$N_d \approx \frac{\ln \frac{T}{R_a}}{\ln(1 + R_r)}$$

Die zu erwartende Beschleunigung  $S$ , die mit diesem Verfahren zu erreichen ist, berechnet sich daher zu:

$$S = \frac{N_n}{N_d} \approx \frac{\frac{T}{R_a} \ln(1 + R_r)}{\ln \frac{T}{R_a}} \approx \frac{\frac{T}{R_a} R_r}{\ln \frac{T}{R_a}}.$$

Wie zu sehen ist, vergrößert sich die beschleunigende Wirkung dieses Verfahrens, je größer das Beobachtungsintervall  $T$  gegenüber der absoluten Zeitauflösung des Simulators  $R_a$  ist und um so geringer die gewünschte relative Genauigkeit  $R_r$  ist.

Sollen beispielsweise während eines 1 ms langen Intervalls Vergleiche durchgeführt werden, sind dazu ca. 700-mal mehr Vergleiche notwendig, wenn eine absolute Genauigkeit von 1 ns gefordert ist, als im Falle einer benötigten, relativen Genauigkeit von zwei Dezimalstellen. Ist das zu untersuchende Intervall 1 Sekunde lang und eine relative Genauigkeit einer Dezimalstelle gefordert, sind sogar ca. 4800000-mal mehr Vergleiche notwendig, wenn die Genauigkeit nicht dynamisch an die Erfordernisse angepaßt wird.

## 5. Modellierungsumgebung VERIFY

Um die VHDL-Erweiterungen aus Abschnitt 2.6, die in Kapitel 3 vorgestellten Methoden zur effizienten Simulation und die Auswertemethoden aus Kapitel 4 testen zu können, wurde die Modellierungsumgebung **VERIFY** (VHDL-based **E**valuation of **R**eliability by **I**njecting **F**aults **e**fficiently) geschaffen.

Einen Überblick über die einzelnen Schritte bei der Bewertung von Fehlertoleranzeigenschaften mit VERIFY gibt Abbildung 49.

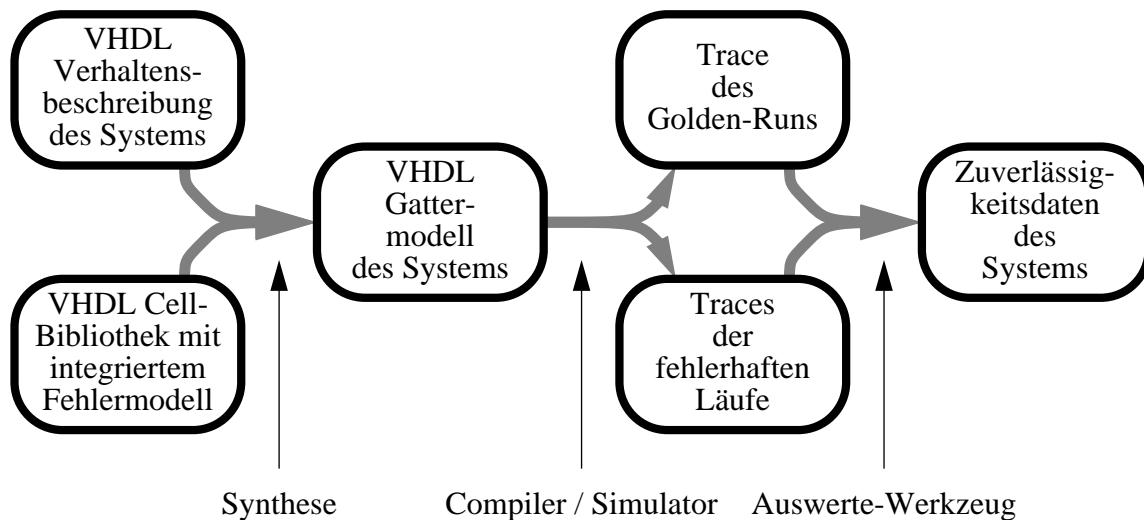


Abb. 49: Modellerstellung, Simulation und Auswertung mit VERIFY

Die Modellierungsumgebung besteht aus einer Reihe verschiedener Werkzeuge, die für die einzelnen Schritte verwendet und in den nachfolgenden Abschnitten 5.1 („Modellerstellung“), 5.2 („Compilierung“), 5.3 („Simulation“) und 5.4 („Auswertung“) kurz beschrieben werden.

Außer dem vorhandenen Synthese-Werkzeug wurden alle Teile der Modellierungs- und Auswertenumgebung in der Programmiersprache „C“ geschrieben. Die lexikalische Analyse sowie das Parsen der Eingabedaten geschieht mit Hilfe von Modulen, die mit den GNU-Werkzeugen `flex` bzw. `bison`<sup>1</sup> generiert wurden. Damit ist es möglich, die Werkzeuge auf allen Plattformen einzusetzen, auf denen die GNU-Werkzeuge `gcc`, `flex` und `bison` laufen. Dies wurde exemplarisch an folgenden Rechnern gezeigt: SUN (SPARC-Prozessor) mit SOLARIS-2 oder SunOS-4, SUN (M68020-Prozessor) mit SunOS-4, DEC (ALPHA-Prozessor) mit OSF-1, INTEL-Prozessoren (I386, I486, PENTIUM) mit LINUX, MOTOROLA-MVME188-Systemen (MEMSY) (M88100-Prozessor) mit MEMSOS oder UNIX-SystemV.

### 5.1 Modellerstellung

Die Modellerstellung erfolgt für alle Experimente nach der in Abbildung 4 vorgestellten Methode. Bei der Modellierung wird jeweils ein Systemmodell auf Verhaltens- oder Register-Transfer-Ebene in der Sprache VHDL erstellt. Es wird nachfolgend in einem ersten Schritt mit Hilfe eines Synthese-Werkzeugs (SYNOPTSYS) in ein Modell auf Gatterebene umgewandelt.

1. Die GNU-Werkzeuge `flex` und `bison` sind verbesserte Versionen der bekannten UNIX-Werkzeuge `lex` bzw. `yacc`.

Dieses Gattermodell kann, da es keinerlei VHDL-Erweiterungen beinhaltet, den üblichen Design-Untersuchungen unterworfen werden, um zum Beispiel versteckte Design-Fehler zu entdecken, die Testbarkeit bezüglich Herstellungsfehlern zu untersuchen usw. Gegebenenfalls kann es durch mögliche Optimierungsstrategien verbessert werden. Wenn das Modell gegebenen Ansprüchen bezüglich Leistung, Testbarkeit usw. genügt, wird es in nachfolgenden Schritten auf seine Verlässlichkeit hin überprüft und gegebenenfalls optimiert.

### 5.2 Compilierung

Nach einer erfolgreichen Durchführung der bei einem Entwurf eines digitalen Systems üblichen Tests kann dann in einem zweiten Schritt die normale, fehlerfreie Zellbibliothek durch eine Bibliothek ersetzt werden, die mit Hilfe der in Kapitel 2.6 vorgestellten VHDL-Erweiterung um Beschreibungen möglicher Fehler ergänzt wurde.

Dieses Modell kann dann vom erstellten Compiler in die Sprache „C“ [Kernighan90] und mit einem beliebigen C-Compiler und Assembler in Maschinensprache übersetzt werden. Es wäre auch möglich, aus dem Quellcode in erweitertem VHDL direkt Maschinen-Code zu generieren. Diese Art der Übersetzung erlaubt es jedoch, den eigentlichen VHDL-Compiler Hardware-unabhängig zu programmieren. Die Hardware-Abhängigkeiten sind bei dieser mehrstufigen Übersetzung vollständig im C-Compiler und Assembler verborgen.

Die einzelnen, compilierten Module werden anschließend mit dem Simulator zu einem ablauffähigen Programm zusammengebunden.

### 5.3 Simulation

Das so entstandene Programm kann gestartet werden. Es simuliert das vorgegebene digitale System. Während des Programmlaufes werden Spuren des fehlerfreien und der einzelnen fehlerbehafteten Läufe geschrieben. Neben den Fähigkeiten eines normalen VHDL-Simulators bietet dieser Simulator die Möglichkeit, die modellierten Fehler zu injizieren. Dabei können sowohl die Zeitpunkte, wann Fehler auftreten sollen, als auch die Orte, an welchen Fehler aktiv sind, eingeschränkt werden. Dadurch können uninteressante Zeitintervalle (z.B. Initialisierungs- oder Terminierungsphase einer Applikation) oder nicht zu betrachtende Komponenten des Systems (z.B. wenn nur eine bestimmte Komponente zu verbessern ist) ausgespart werden. Weiterhin kann die Anzahl der zu injizierenden Fehler und damit die statistische Konfidenz der Ergebnisse beim Start der Simulation vorgegeben werden.

Bei der Simulation werden folgende Daten hierarchisch, der Struktur des Modells entsprechend sortiert, in einer Datenbank abgelegt:

- alle Komponenten
- alle Signale
- alle Fehlermöglichkeiten (mit Auftrittsrate und mittlerer Dauer)

Diese Daten können während der Auswertephase mit in die statistische Auswertung der Experimente einbezogen werden.

Zur Beschleunigung der Simulation werden die in den Abschnitten 3.2 („Einzelfehlerannahme“), 3.4 („Multi-Threaded Fault-Injection“) und 3.6 („Vergleich mit Golden-Run“) beschriebenen Verfahren standardmäßig eingesetzt. Eine Parallelisierung kann durch ein mehrfaches Starten des Programmes auf unterschiedlichen Rechnern erreicht werden (siehe Abschnitt 3.5 „Parallele Simulation“). Die in Abschnitt 3.8 („Irrelevante Fehler in Registern“) und 3.9 („Irre-

levante Fehler in kombinatorischen Schaltungen“) vorgestellten Möglichkeiten zur Simulationsbeschleunigung können verwendet werden, indem nach einer Sensibilitätsüberprüfung durch ein Modell nach Abbildung 40 während eines Golden-Runs die zu injizierenden Fehler entsprechend ausgewählt werden.

Da eine Simulation mit Hilfe des Zielsystems (siehe Abschnitt 3.3) nur im Falle existierender Ziel-Hardware möglich ist, wird diese Beschleunigungsmaßnahme nicht standardmäßig eingesetzt. Ihre Funktionsweise wurde in der Arbeit [Sieh94] demonstriert. Die Möglichkeit, Modelle dynamisch während der Laufzeit zu wechseln (Abschnitt 3.7), wurde anhand eines anderen Werkzeugs gezeigt [Tschäche96a].

Sowohl der Compiler als auch der Simulator sind in der Lage, eine Untermenge der Sprache VHDL sowie die im Abschnitt 2.6 vorgeschlagenen Erweiterungen zu compilieren bzw. zu simulieren. Auf eine vollständige Implementierung von VHDL wurde verzichtet, da nur Beschreibungen digitaler Systeme auf Gatterebene zu verarbeiten sind. Viele Möglichkeiten, die VHDL im Standard von 1993 bietet, sind hierfür nicht erforderlich (z.B. Pointer, Strukturen, Fließkomma-Arithmetik). Der Compiler und Simulator gehen in ihrem Sprachumfang jedoch über das hinaus, was derzeit erhältliche Synthese-Werkzeuge im allgemeinen bieten (z.B. sind `after`-Klauseln und `while`-Schleifen erlaubt).

## 5.4 Auswertung

Die bei der Simulation entstandenen Spuren vom Golden-Run und den Fehler-Simulationsläufen können im Schritt 3 durch ein Auswertewerkzeug automatisch analysiert und dem Benutzer präsentiert werden.

Dazu wird zunächst gemäß den Abschnitten 4.2 und 4.3 ein Vergleich aller fehlerbehafteten Spuren mit dem originalen, fehlerfreien Ablauf durchgeführt. Es ist möglich, vorher bestimmte Signale auszufiltern, die nicht in die Überprüfung mit eingehen sollen. So entsteht eine Datenbank, die eine Liste von Experimentergebnissen enthält. Jeder Eintrag entspricht einem Fehlerinjektionsexperiment und enthält folgende Daten:

- Fehlertyp
- Fehlerort
- Zeitpunkt, wann die Fehlerursache aufgetreten ist
- Zeitpunkt, wann die Fehlerursache wieder verschwunden ist
- Zeitpunkt, wann alle Fehlerauswirkungen wieder verschwunden sind
- Zeitverschiebung gegenüber dem fehlerfreien Lauf

Die in der Datenbank stehenden Daten können dann mit den während der Simulation geschriebenen Daten zusammen gewöhnlichen Statistikoperationen unterworfen und die Ergebnisse angezeigt werden. Beispiele für derartige Auswertungen sind im Abschnitt 6.3 anhand einer größeren Experimentreihe dargestellt.



## 6. Fallbeispiel DP32

Um die Tauglichkeit des Modellierungswerkzeugs **VERIFY** aus Kapitel 5 für die Praxis von Fehlerinjektionsexperimenten zu testen, wurden mehrere Modelle mit Hilfe des Werkzeugs aufgestellt und bezüglich ihrer Fehlertoleranzeigenschaften ausgewertet.

Neben dem Austesten der vorgestellten Modellierungs-, Simulations- und Auswerteverfahren soll gleichzeitig verdeutlicht werden, wie unterschiedlich aufgrund von Modellauswertungen gewonnene Ergebnisse sein können, wenn die in den Abschnitten 2.2, 2.4 und 2.5 aufgezeigten Probleme bei der Modellerstellung nicht korrekt berücksichtigt werden.

Im folgenden Abschnitt 6.1 werden zunächst die verschiedenen, in den Beispielen untersuchten Modelle vorgestellt. Meßwerte bezüglich der Compilierung, Simulation und Auswertung dieser Modelle werden in Abschnitt 6.2 gezeigt. Abschnitt 6.3 präsentiert und diskutiert die durch Simulation und Auswertung gewonnenen Ergebnisse.

### 6.1 Modellerstellung

Für alle Untersuchungen wurde das Modell des DP32-Prozessors aus dem VHDL-Cookbook [Ashenden90] verwendet. Dieses Modell hat den Vorteil, daß es frei verfügbar und vielen Personen bekannt ist. Der modellierte Prozessor ist zwar relativ einfach aufgebaut, übertrifft jedoch in seiner Komplexität die meisten der in der Literatur zu Fehlerinjektionsexperimenten verwendeten Systemmodelle schon um ein Vielfaches. Das Blockschaltbild des Prozessors zeigt Abbildung 50.



```

component adder32
  port (
    res : out std_logic_vector(31 downto 0);
    op1 : in  std_logic_vector(31 downto 0);
    op2 : in  std_logic_vector(31 downto 0));

  add : adder32
    port map ( res => a,
              op1 => b,
              op2 => c);

```

Da das Synthese-Tool nicht in der Lage war, das Multiplikations- und Divisionswerk zu synthetisieren und eine Synthetisierung dieser Komponenten von Hand sehr mühsam und für die angestrebten Ziele unnötig gewesen wäre, wurde auf die Multiplikation und Division verzichtet. Aus denselben Gründen wurde das Register-File von ursprünglich 256 auf 8 Register beschränkt. Zu betonen ist, daß diese Einschränkungen aufgrund von Mängeln des verwendeten Synthese-Tools, das an sich als eines der besten gilt, nicht aber aufgrund von Mängeln des entwickelten Modellierungs- und Fehlerinjektionsverfahrens notwendig wurden. Vollständig synthetisierbare Modelle, die für die Hardware-Herstellung notwendig sind, können ohne jede Änderung zur Zuverlässigkeitsanalyse verwendet werden.

Die Synthese des DP32 zu einem Modell auf Gatterebene wurde mit zwei verschiedenen Zellbibliotheken durchgeführt. Daraus entstanden zwei unterschiedliche Hardware-Modelle des DP32 („Simple“ bzw. „Advanced“).

Die Zellbibliothek für das Modell „Simple“ enthielt die folgenden Komponenten:

- AND-Gatter mit zwei Eingängen
- OR-Gatter mit zwei Eingängen
- Inverter
- 1-Bit-Register
- 1-Bit-Latch
- Tri-State-Treiber

Die Bibliothek für das Modell „Advanced“ verfügte darüber hinaus zusätzlich über die folgenden Komponenten:

- AND-Gatter mit drei bzw. vier Eingängen
- OR-Gatter mit drei bzw. vier Eingängen
- NAND-Gatter mit zwei, drei bzw. vier Eingängen
- NOR-Gatter mit zwei, drei bzw. vier Eingängen
- XOR-Gatter mit zwei, drei bzw. vier Eingängen
- Multiplexer 2-nach-1
- 1-Bit-Volladdierer

Die einzelnen Komponenten der Zellbibliotheken wurden um einfache, bekannte Fehlermodelle erweitert. Sämtliche Gatter und Register enthalten mögliche Stuck-At-0 und Stuck-At-1 Fehler für alle Ein- und Ausgänge (ähnlich Abbildung 10). Tritt einer dieser Fehler auf, wird das Ein- bzw. Ausgangssignal für die Dauer des Fehlers auf ‘0’ bzw. ‘1’ gehalten. Die Fehlerdauer ist exponentiell verteilt mit einer mittleren Länge von einer Taktperiode des Systems (20 ns). Alle Speicherelemente besitzen zusätzlich noch Bit-Flip-Fehler. Im Falle eines derartigen Fehlers kippt der Inhalt der Speicherzelle von ‘0’ nach ‘1’ bzw. von ‘1’ nach ‘0’. Die Gesamt-CPU wur-

## 6. Fallbeispiel DP32

---

de weiterhin noch um mögliche Stuck-At-Fehler an den Ein- und Ausgängen versehen (Pin-Level-Fehler). Alle denkbaren Fehler haben die gleiche Auftrettsrate. Auf diese Weise wurden die drei wohl am weitesten verbreiteten Fehlermodelle implementiert.

Unter Verwendung der einfachen Zellbibliothek ergab sich ein Modell mit 3756 einzelnen Zellen, 4055 internen Signalen und 22792 möglichen Stuck-At-Fehlern. Das Modell „Advanced“ enthält dagegen 2847 Gatter und Register, 3162 interne Signale und 18782 verschiedene Stuck-At-Fehlermöglichkeiten. Beide Modelle besitzen darüber hinaus 419 mögliche Bit-Flip- und 142 Pin-Level-Fehler.

Der Umfang der Erweiterten-VHDL-Quellen des DP32 beträgt 5489 Zeilen erweiterten VHDL-Code für die strukturelle Beschreibung des Prozessors. Die verwendete Zellbibliothek hat einen Umfang von 685 (Modell „Advanced“) bzw. 313 (Modell „Simple“) Zeilen Code. Ohne Fehlerbeschreibungen verringert sich die Anzahl der Code-Zeilen der Bibliothek auf 339 bzw. 229.

Dieser Prozessor wurde um ein Speichermodul und eine Takterzeugungseinheit zu einem vollständigen Hardware-Modell erweitert. In diese Komponenten wurden jedoch keine Fehlermöglichkeiten integriert. Sie werden in den nachfolgenden Ergebnissen jeweils als fehlerfrei angenommen.

Somit standen für Experimente zwei verschiedene Hardware-Modelle mit jeweils integriertem Fehlermodell zur Verfügung. Die wohl bekanntesten Fehlermodelle (Stuck-At-, Bit-Flip- und Pin-Level-Fehler) sind Teilmengen der implementierten Fehlermodelle. Je nach Wahl der verschiedenen Fehlerraten kann damit eine CPU mit überwiegend internen Stuck-At-, Bit-Flip- oder überwiegend Pin-Level-Fehlern dargestellt und simuliert werden.

Die Test-Software wurde ebenfalls aus dem VHDL-Cookbook übernommen. Abbildung 51 zeigt das Programm. Bei Ausführung wird in einer Endlosschleife in einer Speicherzelle (counter) wiederholt ein Wert von 0 nach 9 hochgezählt.

```
initr0
start:   addq(r2, r0, 0)           ! r2 := 0
loop:    sta(r2, counter)         ! counter := r2
         addq(r2, r2, 1)          ! increment r2
         subq(r1, r2, 10)         ! if r2 = 10 then
         brzq(start)              ! restart
         braq(loop)               ! else next loop

counter: data(0)
```

**Abb. 51: Testprogramm für DP32**

Dieses Programm ist zwar sehr einfach gehalten, es zeigt aber bereits unterschiedliches Verhalten gegenüber verschiedenen Fehlertypen (Stuck-At-, Bit-Flip- bzw. Pin-Level-Fehlern) und ist damit für die eingangs erwähnten Ziele dieser Experimente ausreichend. Da bekannt ist (siehe z.B. [Iyer86], [Güthoff95]), daß Ergebnisse von Fehlerinjektionsexperimenten sehr stark von der Last des Systems abhängen, ist für die Bewertung von konkreten Systemen jeweils die entsprechende Last des zu untersuchenden Systems zu verwenden.

## 6.2 Compilierung, Simulation und Auswertung

Die erstellten Modelle des DP32 wurden mit Hilfe des entwickelten Compilers und Simulators (siehe Kapitel 5) zunächst compiliert und nachfolgend simuliert und die dabei entstandenen Spuren analysiert. In diesem Abschnitt sollen Meßwerte präsentiert werden, die etwas über die Zeiten sowie den Ressourcenbedarf (Hauptspeicher sowie Plattenspeicher) der Modellanalyse aussagen. Da die Modellauswertung aufwendig ist, sind diese Zahlen gleichzeitig ein Hinweis darauf, welche Modelle auf einer gegebenen Rechnerkonfiguration noch in einer vertretbaren Zeit mit bei gegebener Hardware-Ausstattung auswertbar sind und welche einen zu großen Rechenaufwand erfordern.

Nach der Modellerstellung laufen alle zur Auswertung des Modells notwendigen Schritte vollautomatisch ab. Die angegebenen Zeiten sind daher reine Maschinen- und keine Ingenieursarbeitszeiten.

Wie in Kapitel 5 beschrieben ist, sind alle Bestandteile der Modellierungsumgebung VERIFY entwickelt worden, um die prinzipielle Machbarkeit dieses Modellierungs- und Simulationsansatzes zu demonstrieren. Ein Tuning der einzelnen Werkzeuge zur Beschleunigung der Experimente wurde nicht vorgenommen. Es wurde besonders auf die zugrundeliegenden Algorithmen, jedoch nicht auf eine besonders effiziente Implementierung Wert gelegt. Es ist daher anzunehmen, daß die Leistungsfähigkeit der Modellierungsumgebung noch um etliches gesteigert werden könnte<sup>1</sup>.

Dieser Abschnitt ist unterteilt in einzelne Unterkapitel, die jeweils einen Schritt bei der Auswertung der Modelle beschreiben. Alle Angaben bezüglich Zeit-, Hauptspeicher- und Plattenspeicherverbrauch beziehen sich auf Messungen auf einem „Digital 2100 Server 500MP“-System von DEC mit ALPHA-CPU (Typ 21064, 190 MHz) und 128 MByte Hauptspeicher. Alle Messungen wurden auch auf SUN-ULTRA-170-Systemen und INTEL-PENTIUM-133-PCs ausgeführt. Da sich die Meßwerte jedoch nur unwesentlich unterscheiden, werden im folgenden nur die Werte für die Messungen auf dem DEC-System vorgestellt.

### Compilierung

Der entwickelte Compiler für die um Fehlerbeschreibungsfähigkeiten erweiterte Sprache VHDL generiert aus den gegebenen Beschreibungen C-Code. Nachfolgend muß daher noch ein C-Compiler, Assembler und Linker eingesetzt werden, um ausführbaren Code zu generieren. Da die Syntax und Semantik der Sprache VHDL nicht wesentlich verändert wurde, ist die Zeit, die für das Compilieren der Sourcen benötigt wird, vergleichbar mit Compilierungszeiten für normale VHDL-Sourcen. So können die vorliegenden „Advanced“- bzw. „Simple“-Modelle in 35 bzw. 46 Sekunden nach C übersetzt werden (zum Vergleich: der VHDL-Compiler von MODELTECH benötigt 29 bzw. 34 Sekunden für die Modelle ohne Fehlerbeschreibungen). Der nachfolgende C-Compiler, der Assembler und Linker benötigen dann jedoch nochmals 3:30 bzw. 3:45 Minuten, um die generierten C-Sourcen zu einem ausführbaren Programm zusammenzusetzen.

Anhand dieses Beispiels zeigt sich, daß gegebene Modelle in relativ kurzer Zeit übersetzt werden können. Die Erweiterungen der Sprache VHDL verlängern die Compilierungszeiten gegenüber reinen VHDL-Sourcen nur unwesentlich.

---

1. Vergleiche mit dem kommerziell erhältlichen VHDL-Compiler und -Simulator der Firma MODELTECH deuten auf einen erreichbaren Beschleunigungsfaktor von etwa 10 hin.

### Simulation

Die Simulationszeiten des „Simple“- bzw. „Advanced“-Modells und der verschiedenen Fehlermodelle unterscheiden sich nur unwesentlich. Daher sind im folgenden nur die Werte für das Advanced-Modell bei internem Stuck-At-Fehlermodell dargestellt. Da die Werte der simulierten Zeiten keine Aussage über die Komplexität der Simulation ermöglichen, wird dafür hier zusätzlich die Anzahl der Ereignisse bei der Simulation angegeben. Im Beispiel des DP32 (Taktfrequenz: 50 MHz) treten etwa  $7 \cdot 10^9$  Ereignisse pro zu simulierender Sekunde auf.

Der Simulator ist in der Lage ca. 200 Ereignisse pro Sekunde zu simulieren. Jedes Ereignis benötigt 16 Byte Plattenspeicherplatz. Alle Signale von Aufzählungstypen – andere werden bei der Simulation von Gattermodellen nicht verwendet – benötigen jeweils ca. 100 Bytes im Hauptspeicher bzw. Swap-Space zur Speicherung; jeder Prozeß ca. 200 Bytes.

Damit ergeben sich für die Simulation des fehlerfreien DP32 von 62µs Länge 381553 Ereignisse. Die zugehörige Simulationszeit beträgt etwa 49 Minuten. Zur Speicherung aller Ereignisse werden etwa 6.1 MByte Plattenspeicher benötigt. Ca. 3.7MByte virtueller Speicher sind zur Speicherung der einzelnen Signale, Zustände und Prozeßdaten notwendig. Da jedoch nur ein kleiner Teil der Prozesse jeweils aktiv ist, muß auch nur ein kleiner Teil der zugehörigen Daten im Hauptspeicher vorhanden sein (ca. 200 KByte). Der Rest kann die meiste Zeit während der Simulation im Swap-Space liegen.

Jede durchgeführte Fehlerinjektion mit einem maximalen Beobachtungsintervall der Länge 2µs bedeutet einen Mehraufwand von durchschnittlich etwa 107 Sekunden Rechenzeit (13500 Ereignisse) und 210 KByte Plattenspeicher. Bei 10000 durchgeführten Fehlerinjektionen summiert sich dies zu etwa 12 Tagen Gesamtrechenzeit (Parallelisierbarkeit nicht ausgenutzt) und 2.1 GByte Plattenplatz. Der Bedarf an virtuellem Speicher verdoppelt sich während einer Fehlerinjektion durch den Einsatz des Verfahrens nach Abschnitt 3.4 und beträgt damit ca. 7.2 MByte.

### Auswertung

Zur Analyse des DP32-Modells war weiterhin eine Auswertung der Simulationsspuren nötig. Um die jeweils protokollierten 10000 einzelnen Spuren zu analysieren und die Ergebnisse in die Datenbank zu schreiben, waren etwa 4 Tage Rechenzeit auf dem DEC-System erforderlich. Während dieser Zeit muß aber kein Ingenieur anwesend sein. Nachfolgend die Datenbank auszulesen und Daten für den Benutzer aufzubereiten, ist je nach erwünschter Auskunft unterschiedlich aufwendig, jedoch meist in wenigen Sekunden durchführbar. Diese Abfragen können daher interaktiv erfolgen.

## 6.3 Ergebnisse

Mit den in Abschnitt 6.1 vorgestellten Modellen standen zwei verschiedene Hardware-Modelle mit jeweils drei verschiedenen Fehlermodellen für Experimente zur Verfügung. Von den sechs Modellkombinationen, die damit möglich waren, wurden vier für die Auswertung ausgewählt. Da beide Hardware-Modelle jeweils die gleichen internen Register und externen Pins verwenden und die Funktion der Modelle die gleiche ist, sind die Auswirkungen von Fehlern in diesen Komponenten (Bit-Flip- bzw. Pin-Level-Fehler) in beiden Hardware-Modellen dieselben. Daher konnten die Kombinationen vom „Simple“- bzw. „Advanced“-Modell mit Bit-Flip-Fehlern bzw. die Kombinationen vom „Simple“- bzw. „Advanced“-Modell mit Pin-Level-Fehlern zu je-

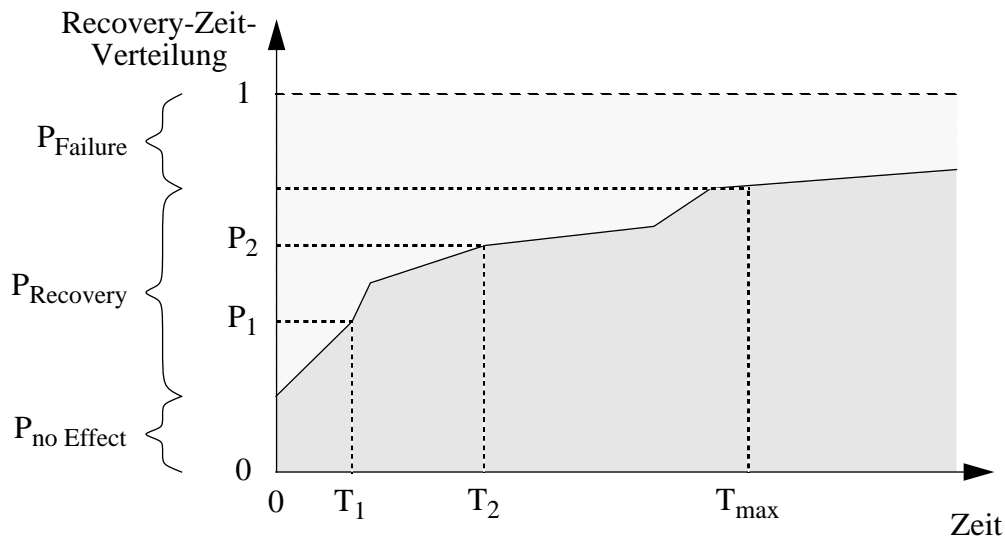
weils einer Kombination zusammengefaßt werden. Der schnelleren Auswertbarkeit wegen wurden die „Advanced“-Modelle bevorzugt. Daraus ergeben sich die vier im folgenden vorgestellten Experimente (Tabelle 7).

**Tabelle 7: Hardware- und Fehlermodell-Kombinationen**

Hardware-Modell	Fehlermodell
„Simple“	Stuck-At
„Advanced“	Stuck-At
„Advanced“	Bit-Flip
„Advanced“	Pin-Level

### 6.3.1 Beschreibung der Diagrammdarstellung

Um möglichst viel Information in einem Diagramm aufzeigen zu können, wurde folgende Darstellungsart gewählt. Auf der X-Achse der Diagramme wird die Zeit nach dem Verschwinden der Fehlerursache abgetragen. In Richtung der Y-Achse ist die Wahrscheinlichkeit  $P(t)$  aufgetragen, daß die Auswirkungen eines aufgetretenen Fehlers nach einer Zeit kleiner oder gleich  $t$  wieder behoben sind. Abbildung 52 zeigt ein Beispiel.



**Abb. 52: Beispieldiagramm**

Die Wahrscheinlichkeit, daß Auswirkungen eines Fehlers innerhalb von maximal  $T_1$  Zeiteinheiten beseitigt werden können, ist  $P_1$ . Notwendige Recovery-Maßnahmen von bis zu  $T_2$  Zeiteinheiten Dauer sind nach dem Auftreten eines Fehlers mit einer Wahrscheinlichkeit von  $P_2$  zu erwarten usw.

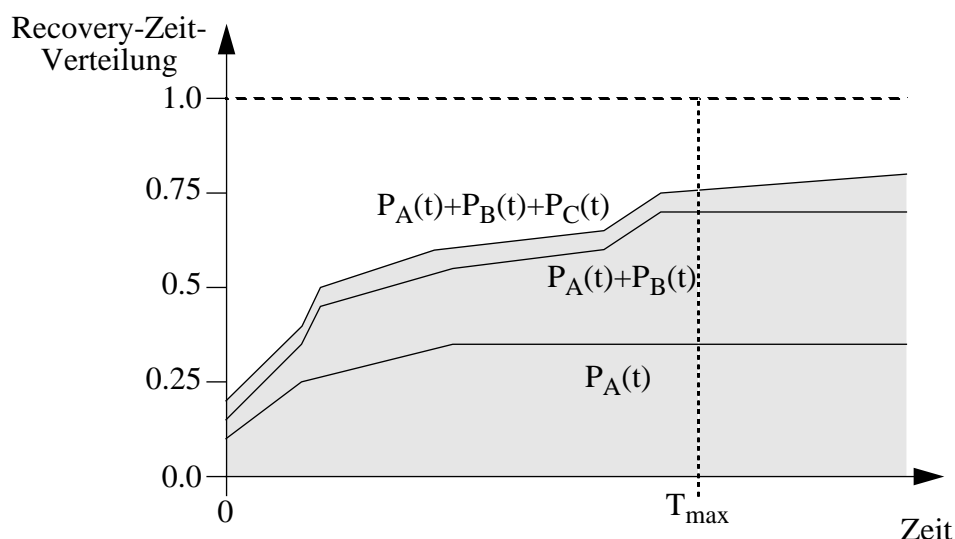
Aus dieser Art Diagramm lassen sich auf einfache Weise eine Vielzahl interessanter Daten ersehen. Anhand der Größe  $P_{Failure}$  läßt sich die Wahrscheinlichkeit ablesen, mit der auftretende Fehler zu Systemausfällen (Fehler mit einer Zeitspanne größer als eine vom Designer vorgegebene Höchstdauer  $T_{max}$ ) führen ( $P_{Failure} = 1 - P(T_{max})$ ). Mit der Wahrscheinlichkeit  $P_{Recovery} = P(T_{max}) - P(0)$  führt ein auftretender Fehler zu einer erfolgreichen Recovery-Aktion.  $P_{noEffect} = P(0)$  zeigt an, wie groß die Wahrscheinlichkeit ist, daß ein auftretender Fehler

## 6. Fallbeispiel DP32

keinerlei Auswirkungen nach sich zieht. Die Diagramme können noch weiter mit Informationen gefüllt werden, wenn die Werte  $P(t)$  nach Fehlern in den einzelnen Subkomponenten des Systems unterteilt werden:

$$P(t) = \sum_{k \in K} P_k(t)$$

Dabei sei  $K$  die Menge der Subkomponenten des Systems und  $P_k(t)$  die Wahrscheinlichkeit der in weniger als  $t$  Zeiteinheiten abgeschlossenen Recovery-Vorgänge, die durch Fehler in der Subkomponente  $k$  ausgelöst wurden. Das heißt, daß der dunkelgrau hinterlegte Bereich der Abbildung 52 untergliedert wird. Abbildung 53 zeigt hierzu ein Beispiel.



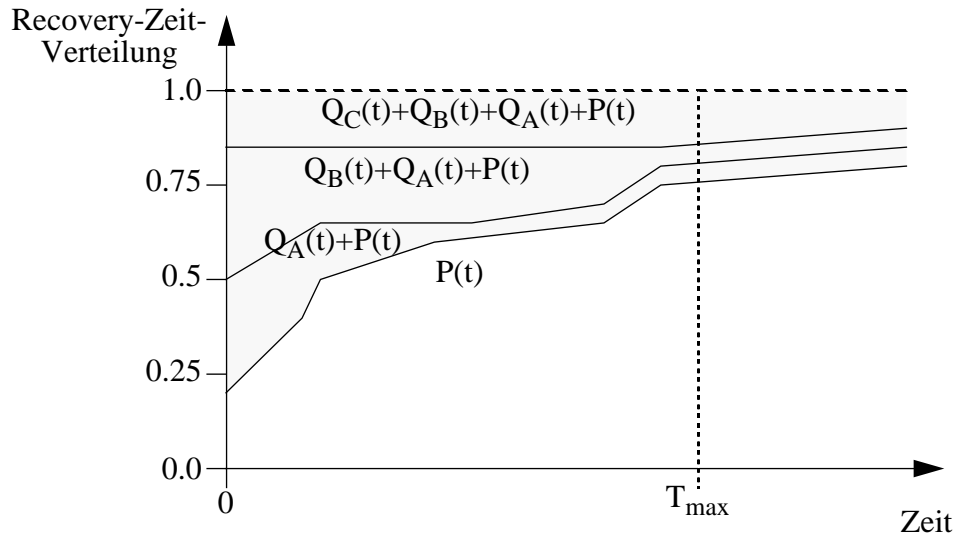
**Abb. 53: Unterteilung der Recovery-Zeiten nach Fehlerorten (bereits behobene Fehler)**

Das Beispiel zeigt eine Unterteilung der Funktion  $P(t)$  nach drei Komponenten ( $A$ ,  $B$  und  $C$ ). Es ist in diesem Beispiel zu sehen, daß die Fehler in der Komponente  $A$  für den Großteil der kurzen, Fehler in Komponente  $B$  für die längeren Recovery-Zeiten verantwortlich sind. Fehler, die in Komponente  $C$  auftreten, führen nicht zu Recovery-Vorgängen.

Im obigen Beispiel kann man gut ersehen, nach welchen Zeiten Fehler welcher Komponenten korrigiert werden. Zur Modellanalyse ist die Information, welche Fehler nach bestimmten Zeiten noch nicht behoben wurden, häufig genauso wichtig. Daher sollte auch diese Information in entsprechenden Diagrammen dargestellt werden. Dies entspricht einer Untergliederung des hellgrau hinterlegten Bereichs in Abbildung 52. In Richtung der Y-Achse wird jetzt die Wahrscheinlichkeit  $Q(t)$  abgetragen, daß die Auswirkungen eines aufgetretenen Fehlers nach einer Zeit kleiner oder gleich  $t$  noch nicht wieder behoben sind. Es gilt  $Q(t) = 1 - P(t)$  und für alle Komponenten  $k$ :  $P_k(t) + Q_k(t) = const$  mit

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

Abbildung 54 zeigt ein Beispiel für eine Unterteilung nach Auftrittsorten bisher noch nicht korrigierter Fehler.



**Abb. 54: Unterteilung der Recovery-Zeiten nach Fehlerorten (noch zu korrigierende Fehler)**

In diesem Beispiel wird die Fläche über dem Graphen, der die Verteilung der Recovery-Zeiten repräsentiert, in einzelne Gruppen unterteilt, die noch zu korrigierende Fehler einzelner Subkomponenten des Systems beschreiben. Anhand dieses Diagramms ist z.B. zu erkennen, daß die meisten zu korrigierenden Fehler in den Komponenten A und B auftreten. Innerhalb der Komponente A ereignen sich etwa 30% ( $Q_A(0) \approx 0.3$ ), in B ca. 35% ( $Q_B(0) \approx 0.35$ ) und innerhalb von C etwa 15% aller Fehler, die korrigiert werden müssen ( $Q_C(0) \approx 0.15$ ). Die verbleibenden 20% der injizierten Fehler haben keinerlei Auswirkungen ( $P_A(0) + P_B(0) + P_C(0) \approx 0.2$ ). Im Beispiel ist die Korrektur der meisten Fehler in der Komponente A etwa nach  $T_{max}/2$  Zeiteinheiten abgeschlossen (der Wert von  $Q_A(t)$  sinkt auf ca. 5% für  $t = T_{max}/2$ ). Die Fehler der Komponente B werden meistens vor der kritischen Zeit  $T_{max}$  berichtigt ( $Q_B(T_{max}) < 0.05$ ), während viele Fehler in der Komponente C erst nach mehr als  $T_{max}$  Zeiteinheiten berichtigt werden ( $Q_C(t)$  ist nahezu konstant für  $t \in [0, T_{max}]$  und nimmt erst für Werte von  $t$  größer  $T_{max}$  langsam ab).

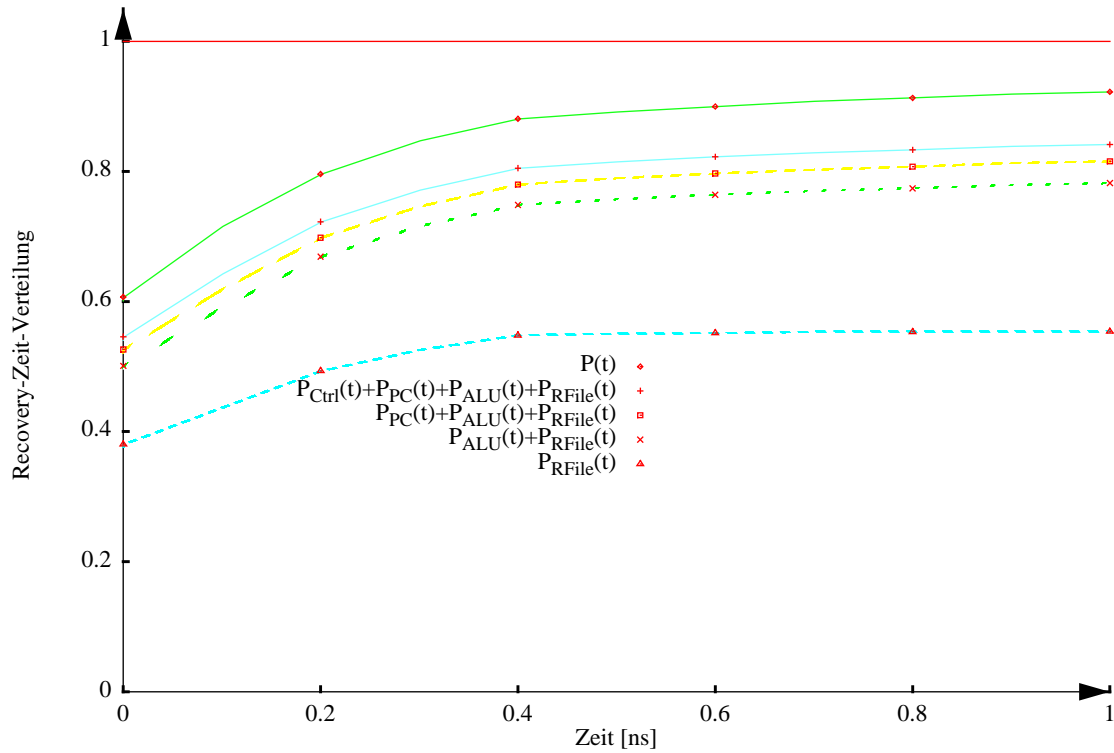
### 6.3.2 Ergebnisse

Die folgenden Abbildungen zeigen die gemessenen Zuverlässigkeitswerte der verschiedenen Hardware- und Fehlermodell-Kombinationen des Beispielsystems DP32. Jede Hardware- und Fehlermodell-Kombination wurde mit je 10000 Einzelfehlern durchgetestet. Die Recovery-Zeit-Verteilungen werden in je zwei Diagrammen dargestellt. Das jeweils erste zeigt die bis zu einem bestimmten Zeitpunkt bereits korrigierten, das zweite die noch zu korrigierenden Fehler der einzelnen Komponenten gemäß den Abbildungen 53 und 54. Eine Ausnahme bildet das Diagramm zur Darstellung der Systemreaktion auf Pin-Level-Fehler (Abbildung 61). In diesem Fall wurden Fehler außerhalb der CPU injiziert. Damit ist in dem einen Diagramm keine Unterteilung nach CPU-Komponenten möglich und auch nicht notwendig.

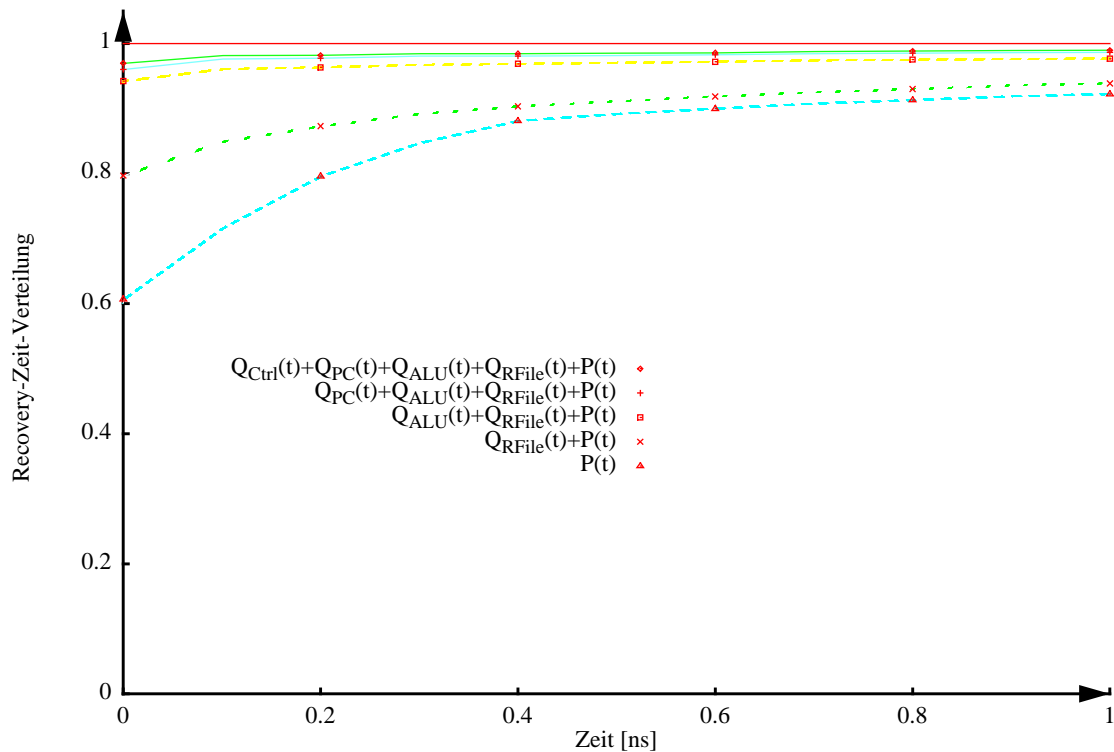
#### „Simple“-Modell mit internen Stuck-At-Fehlern

Die Abbildungen 55 und 56 beschreiben das Verhalten des einfachen Hardware-Modells („Simple“) unter dem Einfluß von internen Stuck-At-Fehlern.

## 6. Fallbeispiel DP32



**Abb. 55: Verteilung der Recovery-Zeiten des „Simple“-Modells bei Stuck-At-Fehlern mit Unterteilung der bereits korrigierten Fehler**



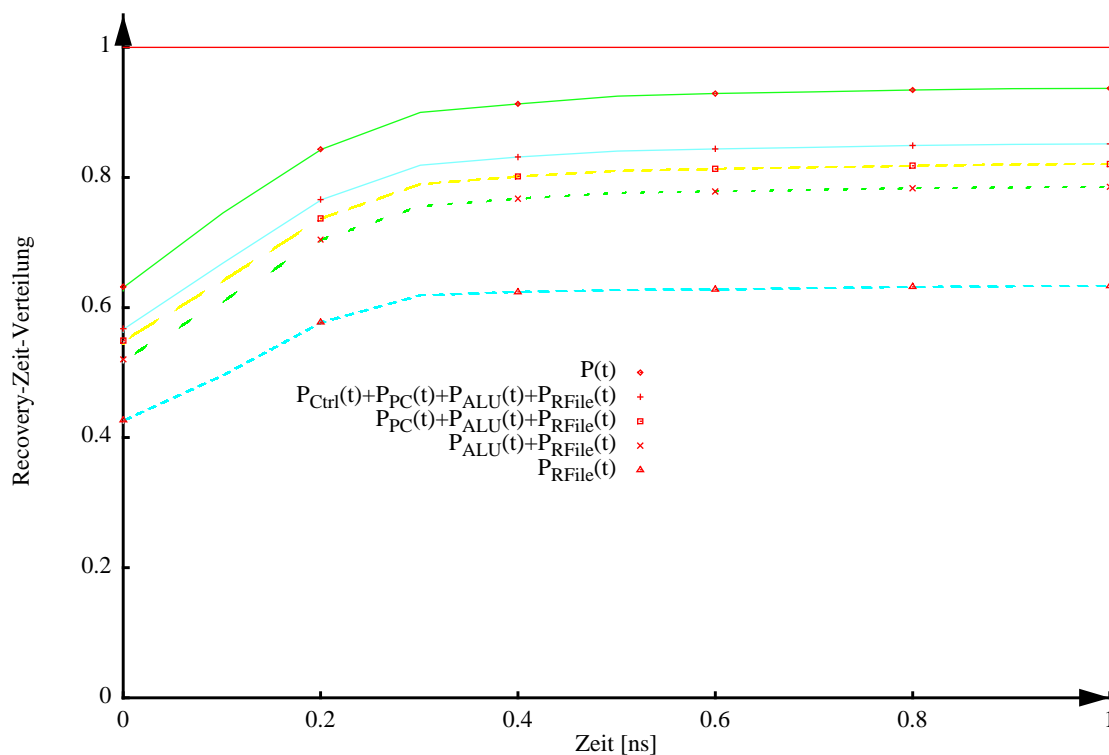
**Abb. 56: Verteilung der Recovery-Zeiten des „Simple“-Modells bei Stuck-At-Fehlern mit Unterteilung der noch zu korrigierenden Fehler**

Es ist zu sehen, daß ein großer Teil (ca. 60%) der injizierten Fehler nach dem Verschwinden der Fehlerursache keinerlei Auswirkungen hat. Weitere ca. 30% der Fehler sind nach 1ns Zeit wieder aus dem System verschwunden. So bleiben nur ca. 10% aller injizierten Fehler, die nicht innerhalb der ersten Nanosekunde vom System korrigiert werden. Von den überhaupt auftretenden Fehlern entfallen etwa 2/3 auf das Register-File, das im Falle des DP32 den größten Teil der Einzelkomponenten umfaßt. Die meisten Fehler, die nicht nach einer Nanosekunde behoben sind, treten jedoch in der ALU auf. Sollen die Fehlertoleranzeigenschaften dieses Systems gegenüber temporären Stuck-At-Fehlern verbessert werden, wäre daher eine Modifikation der ALU zu empfehlen.

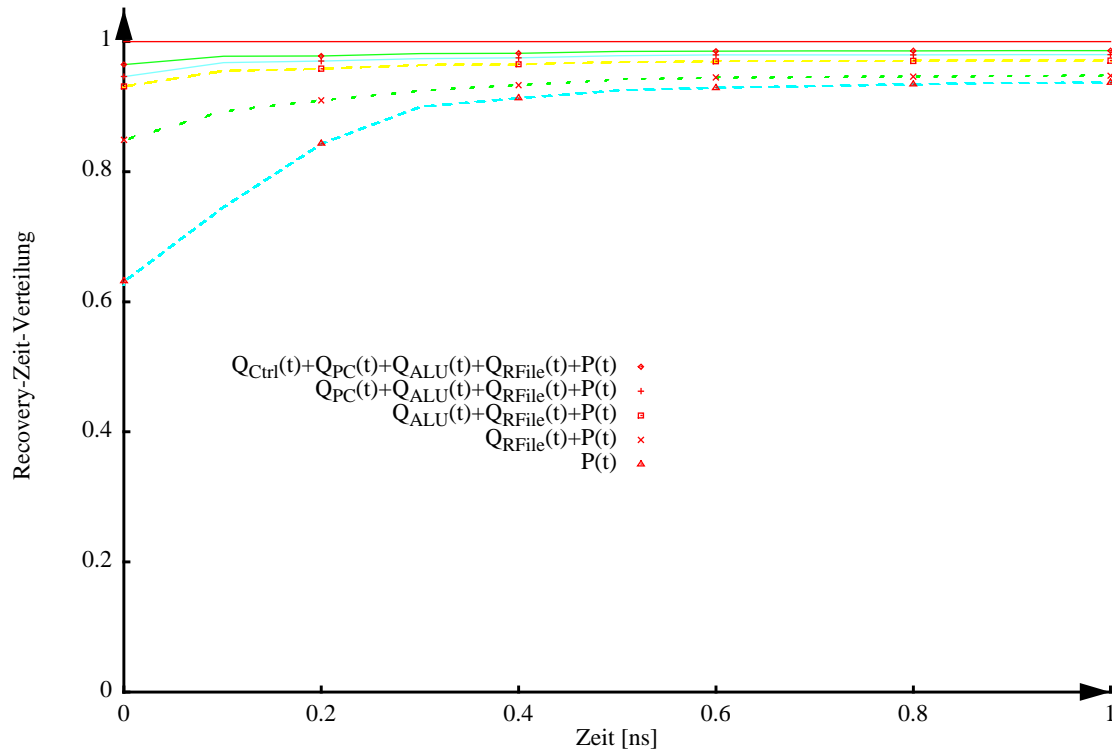
### „Advanced“-Modell mit internen Stuck-At-Fehlern

Die folgenden Abbildungen (Abbildung 57 bis Abbildung 61) geben Meßwerte wieder, die durch Verwendung des „Advanced“-Hardware-Modells gewonnen wurden.

Die Abbildungen 57 und 58 beschreiben das Verhalten dieses Modells unter dem Einfluß von internen Stuck-At-Fehlern.



**Abb. 57: Verteilung der Recovery-Zeiten des „Advanced“-Modells bei Stuck-At-Fehlern mit Unterteilung der bereits korrigierten Fehler**

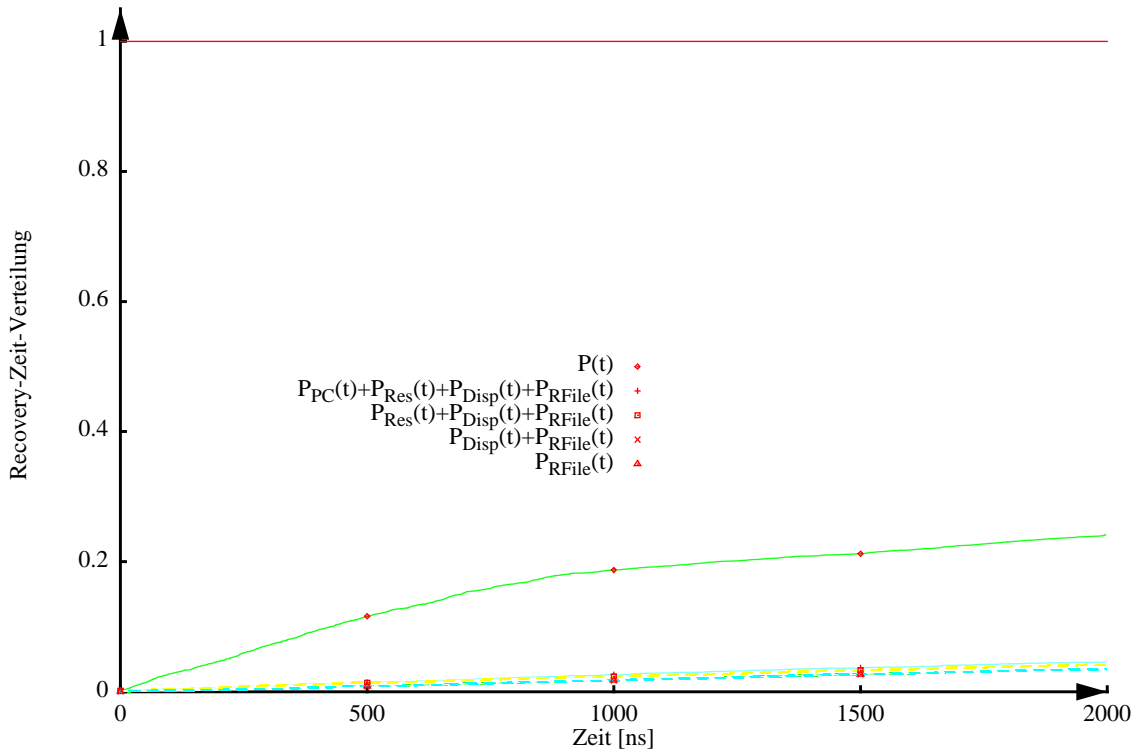


**Abb. 58: Verteilung der Recovery-Zeiten des „Advanced“-Modells bei Stuck-At-Fehlern mit Unterteilung der noch zu korrigierenden Fehler**

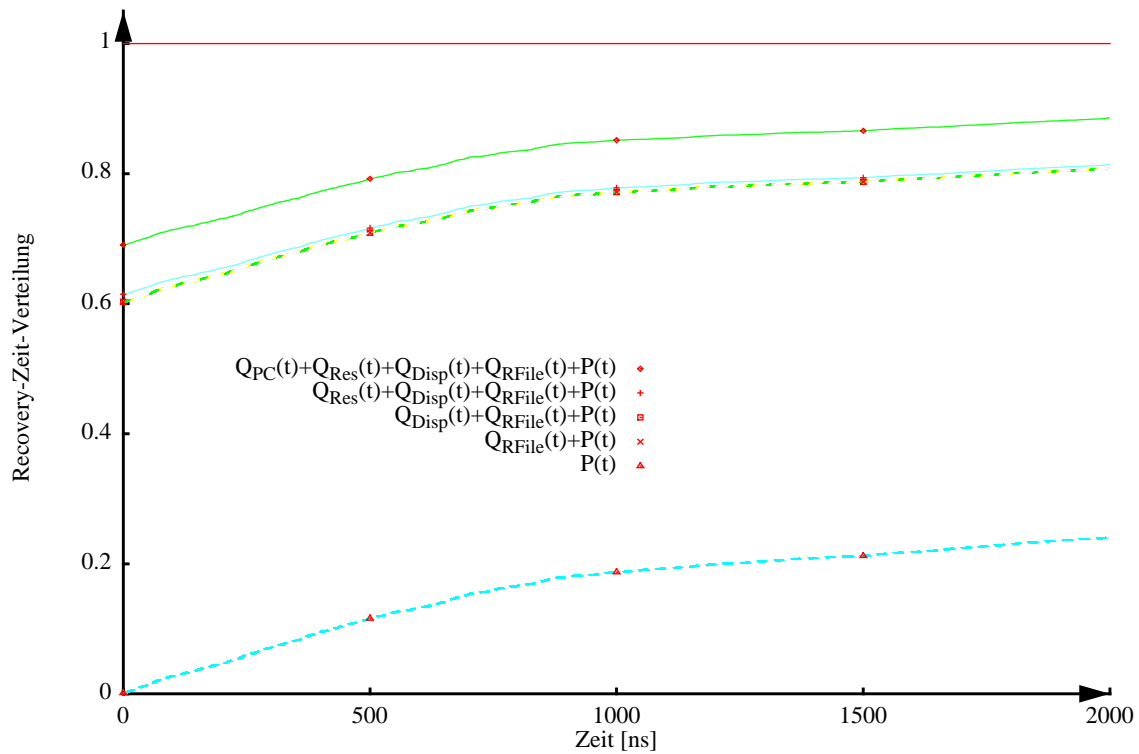
Deutlich ist eine Ähnlichkeit des Recovery-Verhaltens des „Simple“- und des „Advanced“-Modells bei internen Stuck-At-Fehlern zu erkennen. Es ist aber auch zu sehen, daß das „Advanced“-Modell insgesamt mit einer etwas höheren Wahrscheinlichkeit Fehler toleriert ( $P(1ns) = 93.7\%$  beim „Advanced“-Modell bzw.  $P(1ns) = 92.2\%$  beim Modell „Simple“). Bezieht man zusätzlich die Tatsache mit in die Überlegungen ein, daß das „Advanced“-Modell weniger Komponenten und damit weniger Fehlermöglichkeiten (19201 gegenüber 23211) besitzt, ergibt sich eine deutlich geringere Ausfallrate des „Advanced“-Modells.

### „Advanced“-Modell mit internen Bit-Flip-Fehlern

Bit-Flip-Fehler provozieren ein anderes Recovery-Verhalten als Stuck-At-Fehler. Ihre Auswirkungen auf das Verhalten des Systemmodells ist in den Diagrammen in den Abbildungen 59 und 60 gezeigt.



**Abb. 59: Verteilung der Recovery-Zeiten des Modells bei Bit-Flip-Fehlern mit Unterteilung der bereits korrigierten Fehler**



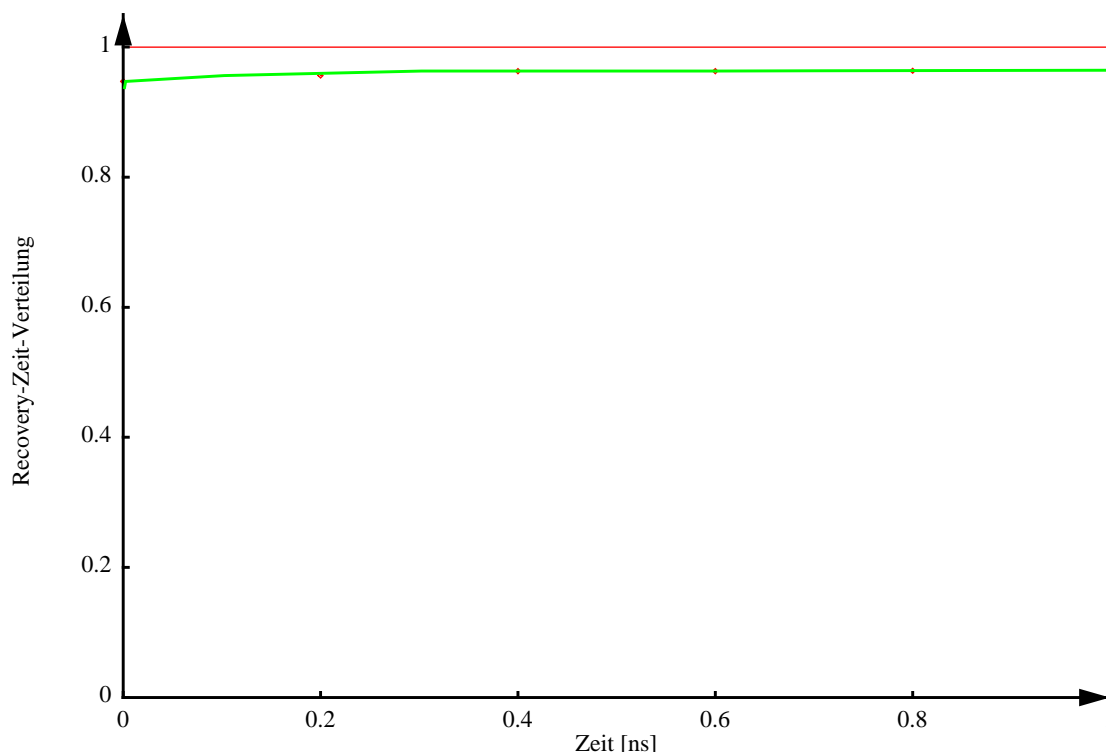
**Abb. 60: Verteilung der Recovery-Zeiten des Modells bei Bit-Flip-Fehlern mit Unterteilung der noch zu korrigierenden Fehler**

## 6. Fallbeispiel DP32

Aus den Diagrammen geht hervor, daß Bit-Fehler im allgemeinen eine deutlich längere Recovery-Zeit benötigen als interne Stuck-At-Fehler (man beachte den anderen Zeitmaßstab!). Auch sind die Fehler innerhalb der Komponenten der CPU anders verteilt. Zwar treten im Register-File weiterhin die meisten Fehler auf, z.B. wird die ALU jedoch nicht von injizierten Fehlern betroffen, da sie (im Falle des DP32) keinerlei Register enthält. Unter den Bit-Fehlern sind in diesem Beispiel die Fehler im Register-File sowie diejenigen, die den Program-Counter (PC) betreffen, besonders zu beachten, da sie nur sehr langsam korrigiert werden können (94% der Bit-Fehler im Register-File und im PC sind nach  $2\mu\text{s}$  noch nicht behoben).

### „Advanced“-Modell mit Pin-Level-Fehlern

Die von den bekannten Pin-Level-Fehlerinjektoren simulierbaren Pin-Level-Fehler erzeugen eine Recovery-Zeit-Verteilung wie in Abbildung 61 gezeigt.



**Abb. 61: Verteilung der Recovery-Zeiten bei Pin-Level-Fehlern**

94.7% aller Pin-Level-Fehler haben keinerlei Einfluß auf das Verhalten des DP32. Diese Tatsache ist damit zu erklären, daß der DP32-RISC-Prozessor nur selten Daten mit dem Hauptspeicher austauscht, da er im wesentlichen auf den Daten seines internen Register-Files arbeitet. Es ist jedoch auch zu sehen, daß die verbleibenden 5.3% der Fehler nur sehr langsam bzw. nicht korrigiert werden können. So sind auch nach  $2\mu\text{s}$  noch 3.0% der Fehler nicht vollständig aus dem System verschwunden. Dies zeigt, daß die Daten, die zwischen dem Hauptspeicher und der CPU ausgetauscht werden, praktisch immer für den Fortgang der Rechnung wesentlich sind.

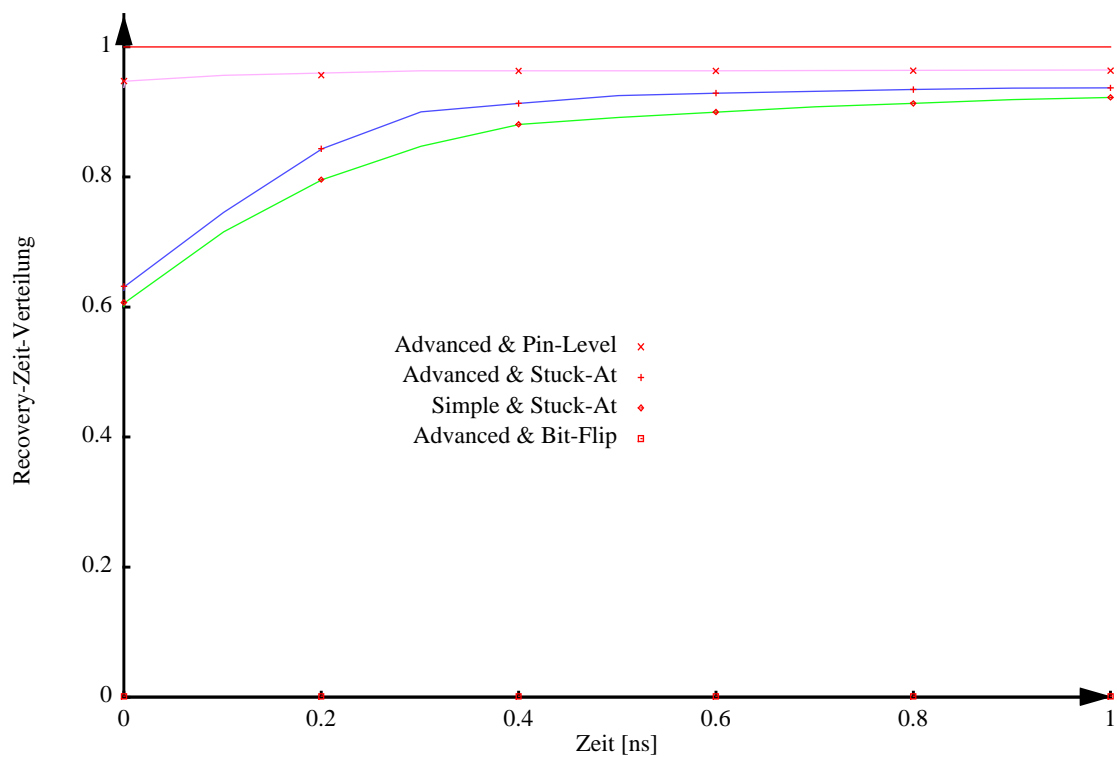
### Vergleich der Ergebnisse

Die beiden folgenden Diagramme (Abbildung 62 und 63) vergleichen die verschiedenen Hardware- und Fehlermodelle miteinander. Beide Diagramme zeigen dieselben Meßkurven, jedoch mit unterschiedlichem Zeitbereich. Es ist aber zu beachten, daß nur die Recovery-Zeit-Verteilungen dargestellt sind. Um die Systeme vollständig miteinander vergleichen zu können, muß

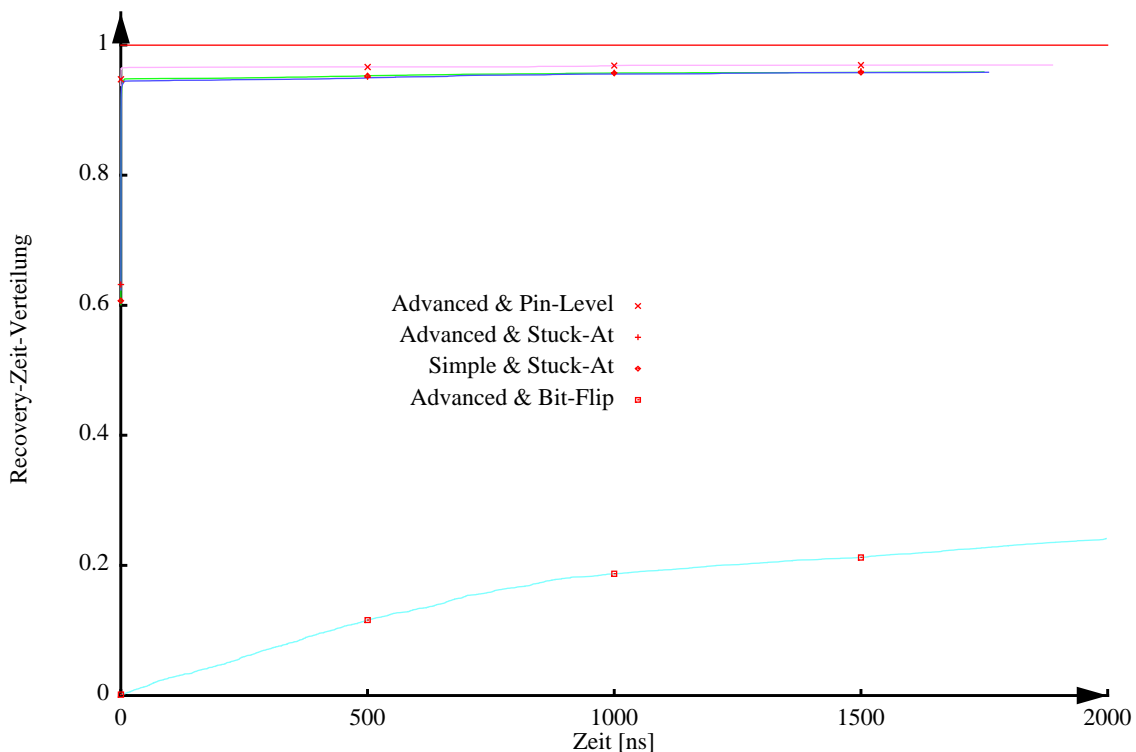
die Fehlerrate der einzelnen Systeme mit in den Vergleich einbezogen werden. Da die Fehler-rate sehr von der verwendeten Technologie und den Bedingungen des Umfeldes abhängt, unter denen ein System arbeitet, sei hier nur die Anzahl der verschiedenen Fehlermöglichkeiten genannt (siehe Tabelle 8).

**Tabelle 8: Anzahl der verschiedenen Fehlermöglichkeiten**

Hardware-Modell	Fehlermodell	Anzahl der verschiedenen Fehlermöglichkeiten
„Simple“	temporäre Stuck-At-Fehler	22792
„Advanced“	temporäre Stuck-At-Fehler	18782
„Advanced“	Bit-Flip-Fehler	419
„Advanced“	temporäre Pin-Level-Fehler	142



**Abb. 62: Recovery-Zeit der verschiedenen Hardware- und Fehlermodelle (0-1ns)**



**Abb. 63: Recovery-Zeit der verschiedenen Hardware- und Fehlermodelle (0-2000ns)**

Die Diagramme zeigen, daß die Verteilungen der Recovery-Zeiten qualitativ unterschiedlich sind, wenn unterschiedliche Fehlermodelle verwendet werden. Die Beispiele machen deutlich, daß Stuck-At-Fehler und Pin-Level-Fehler weitgehend Auswirkungen zeigen, die in kurzer Zeit wieder behoben werden können, während Bit-Flip-Fehler eine sehr viel längere Recovery-Zeit benötigen. Solange die Fehlerraten der einzelnen Fehler nicht bekannt sind, ist daher keine Aussage darüber zu treffen, mit welchen Recovery-Zeiten im Systemverhalten zu rechnen ist. Überwiegen Bit-Flip-Fehler im DP32-System, werden hauptsächlich mittlere Recovery-Zeiten über 1000ns zu erwarten sein, im Falle vermehrt auftretender Stuck-At- bzw. Pin-Level-Fehlern dagegen mittlere Recovery-Zeiten unter 1ns.

In der Literatur angegebene Ergebnisse von Vergleichen von Systemen (z.B. [Hönig94], [Steininger97]), die auf bestimmten Fehlerinjektionsverfahren und damit auf bestimmten, aber in den angegebenen Fällen nicht validierten Fehlermodellen beruhen, sind daher sehr zweifelhaft.

### Zusammenfassung

Für den Hardware-Designer, der einen Prozessor wie den DP32 bezüglich seines Verhaltens gegenüber Fehlern verbessern will, ergeben sich aus den oben vorgestellten Ergebnissen viele Hinweise, wo zweckmäßigerweise mit der Optimierung begonnen werden sollte. Um die Wahrscheinlichkeit zu verringern, daß der Prozessor aufgrund von Fehlern ausfällt, sind alle diejenigen Komponenten zu verbessern, deren Recovery-Zeiten im Mittel überdurchschnittlich lang sind bzw. deren Fehler zu Ausfällen führen. Sowohl beim „Advanced“- als auch beim „Simple“-Modell ist dies im Falle interner Stuck-At-Fehler in erster Linie die ALU. Dies läßt sich dadurch erklären, daß die ALU zur Durchführung von Anwendungsbefehlen (Addition, Subtraktion, usw.), für den Transport von Daten aus dem Register-File zum Hauptspeicher und auch zum Weiterschalten des PC und damit für den korrekten Kontrollfluß benötigt wird.

Im Falle von vermehrt auftretenden Bit-Flip-Fehlern müssen besonders der PC und das Register-File abgesichert werden (z.B. durch Parity-Bits oder eine Error-Correcting-Code-Logik). Wie die Diagramme 59 und 60 zeigen, führen Bit-Flip-Fehler in diesen Komponenten im allgemeinen zu langen Recovery-Zeiten. Dies ist verständlich, wenn man sich vor Augen führt, daß Daten, die sich in diesen Komponenten befinden, fast immer für den weiteren Fortgang der Berechnung weiterverwendet werden. Dies ist bei anderen Registern wie z.B. dem Addr-Register oder dem Res-Register nicht der Fall. Die Daten in diesen Registern werden häufig vor einer weiteren Verwendung wieder neu gesetzt.

Neben dem Hardware-Designer bekommt auch der Software-Entwickler Hinweise, wie seine Programme zur Verbesserung der Fehlertoleranz des Systems zu ändern sind. Treten – wie in diesem Beispiel – interne Stuck-At-Fehler besonders häufig in der ALU auf, so sollten für den Fortgang der Berechnung besonders kritische Werte mehrfach berechnet und die Ergebnisse miteinander verglichen werden. Wenn jedoch bekannt ist, daß überwiegend Bit-Flip-Fehler auftreten, sollten wichtige Daten nicht lange in den General-Purpose-Registern gehalten, sondern nach gewissen, den Anforderungen an das System entsprechenden Zeiten neu berechnet werden.



# 7. Zusammenfassung und Ausblick

## 7.1 Zusammenfassung

Mit der vorliegenden Arbeit konnten Methoden entwickelt werden, die es erlauben, Computersysteme einschließlich ihrer möglichen permanenten und temporären Fehler auf einfache Weise zu modellieren. Die Modelle können mit den vorgestellten Methoden sehr effizient und detailliert simuliert werden. Während der Simulation werden zahlreiche unterschiedliche Fehler-szenarien durchgetestet. Die Simulation und die Auswertung der Spuren kann vollautomatisch durchgeführt werden. Es ist nach dem Aufstellen der Modelle keinerlei weitere Ingenieursarbeit notwendig.

Die vorgestellten Methoden ermöglichen es, daß für die Zuverlässigkeitsbewertung von Systemen detaillierte Modelle auf einfache Weise aufgestellt werden können, die sämtliche Informationen für eine automatische Auswertung enthalten. Die Modellierung orientiert sich an den bekannten Modellierungssprachen wie z.B. VHDL oder VERILOG. Durch kleine syntaktische, aber starke semantische Erweiterungen ist es möglich, die Beschreibung der Hardware und ihrer möglichen Fehler zu einem einzigen Modell zusammenzufassen, wodurch sich eine Reihe von Vorteilen ergibt.

Durch die Vereinigung der Beschreibungen des fehlerfreien und des fehlerhaften Verhaltens von Komponenten in einem einzigen Text wird die Möglichkeit von Inkonsistenzen innerhalb der Beschreibungen drastisch verringert. Es ist in diesem Ansatz praktisch unmöglich, versehentlich ein unpassendes Fehlermodell zu einer Komponente hinzuzufügen. Das Fehlermodell muß beim Auswechseln von einzelnen Komponenten nicht extra modifiziert werden. Es wird mit den Komponenten zusammen ausgetauscht. Da die Modelle für die Zuverlässigkeitsüberprüfung Produkte des normalen Design-Prozesses einer digitalen Schaltung sind und nicht speziell für die Überprüfung erstellt werden, ist eine Korrespondenz zwischen Modell und Realität besser gegeben als in vielen, in der Literatur angegebenen Modellierungsverfahren. Zusätzlich enthält ein Modell alle Fehlerparameter wie z.B. Häufigkeit des Fehlers und Verteilung der Fehlerdauer. Somit kann eine Auswertung eines solchen Modells ohne zusätzliche Informationen durchgeführt werden. Die Modelle sind damit vollständig.

Um solche Modelle auszuwerten, sind im Normalfall sehr viele Simulationsläufe erforderlich. Ziel dieser Arbeit war es daher, Methoden zu entwickeln, um derartige Simulationen möglichst effizient durchführen zu können. Je nach Art der zu simulierenden Fehler und der zu erwartenden Auswirkungen konnten Techniken vorgestellt werden, welche die Gemeinsamkeiten der einzelnen Fehlersimulationen ausnutzen und damit die Simulationen selbst beträchtlich beschleunigend. So ist es möglich, derartige Simulationen bei Anwendung der vorgestellten Techniken um mehrere Größenordnungen schneller abzuwickeln. Für alle Methoden konnten die Voraussetzungen erarbeitet werden, die für ihre Anwendung notwendig sind. Ebenso wurde detailliert auf die Kosten sowie den Nutzen der einzelnen Verfahren eingegangen.

In den meisten Veröffentlichungen, die sich mit temporären Fehlern beschäftigen, wird nicht auf die eigentliche Auswertung der vorgestellten Experimente eingegangen, obwohl auch die Qualität der Auswertung natürlich große Auswirkungen auf die Ergebnisse der Experimente hat. Daher wurde in der vorliegenden Arbeit die Genauigkeit der Auswertung besonders berücksichtigt. Es konnten Verfahren erarbeitet werden, die Spuren von Fehlerinjektionsexperimenten vollautomatisch aufzeichnen und auswerten. Im Gegensatz zu bekannten Verfahren aus der Li-

## 7. Zusammenfassung und Ausblick

---

teratur ist es so z.B. auch möglich, ein aufgrund von Fehlern aufgetretenes, zeitverschobenes Verhalten des Systems richtig zu klassifizieren (Abschnitt 4.2). Anhand von Beispielen wurde der Nutzen dieses Ansatzes gezeigt.

Alle vorgestellten Methoden konnten in der Praxis erprobt werden. Dazu wurde das Modellierungs- und Auswertewerkzeug VERIFY erstellt. Mit dessen Hilfe konnten größere, aus der Literatur bekannte Modelle auf einfache Weise um Fehlermodelle erweitert und ihr Fehlerverhalten bezüglich verschiedener Fehlertypen analysiert werden. Die Ergebnisse werden präsentiert und diskutiert. Damit konnte gleichzeitig ein Vergleich von verschiedenen Fehlermodellen vorgenommen werden.

Das spezifizierte und implementierte Werkzeug wird zur Zeit für Zuverlässigkeitsanalysen von verschiedenen Systemen verwendet. Die vorgestellten Methoden werden in bekannte Fehlerinjektionswerkzeuge integriert.

### 7.2 Ausblick

Trotz einer möglichen Beschleunigung der Fehlerinjektionsexperimente um mehrere Größenordnungen ist es bisher nicht möglich, eine Zuverlässigkeitsauswertung von größeren Systemen sehr schnell und damit interaktiv durchzuführen. Durch die vorgestellten Methoden können zwar auch praktisch relevante Systeme innerhalb akzeptabler Zeiten analysiert werden, die dafür benötigten Rechenzeiten und Speicherressourcen sind jedoch immer noch sehr hoch. Eine geschickte Implementierung der einzelnen Verfahren könnte hier nochmals eine geschätzte Beschleunigung um einen Faktor von etwa 10 und einen geringeren Speicherplatzbedarf bewirken.

Alle vorgestellten Verfahren wurden für digitale Computersysteme entwickelt und an ihnen erprobt. Dort haben sie sich bewährt. Nicht geklärt werden konnte, wie Fehler in analogen Systemen zu modellieren und zu simulieren sind. Im Falle analoger Störungen treten noch zusätzliche Parameter wie z.B. die Stärke der Störung oder die Form der Störung auf. Bei der Auswertung stellt sich die Frage, wie die Abweichung im Verhalten des fehlerhaften Systems vom Verhalten des fehlerfreien Systems gemessen werden kann.

Da die Anzahl der gemischt-analog-digitalen Systeme z.B. in Form von Embedded-Controllern immer größer wird und der analoge Teil häufig eine wichtige Rolle in diesen Systemen spielt, wäre eine Einbeziehung der Fehler im analogen Teil der Systeme für eine Fehlertoleranzbewertung sehr nützlich.

Derartige analoge Fehlermodelle könnten ähnlich zur vorliegenden Arbeit z.B. in die Modellierungssprache Analog-VHDL integriert und getestet werden.

In dieser Untersuchung ging es darum, die Spuren der einzelnen Fehlerinjektionsexperimente anhand der Spuren des Golden-Runs zu klassifizieren. Wie die Abbildungen 41 und 42 zeigen, existieren jedoch möglicherweise andere Pfade, die auch zu einem nach außen korrekt wirkenden Systemverhalten führen. Die Information über derartige Pfade kann jedoch nicht dem Golden-Run entnommen werden. Hierfür sind andere Mechanismen erforderlich. Um die aufgezzeichneten Verhalten des Systems unter Fehlern vollständig klassifizieren zu können, sind erkennende Automaten notwendig, welche die Menge der erlaubten Pfade von der Menge der nicht erlaubten unterscheiden können. Zur Programmierung dieser Automaten sind neuartige Beschreibungsmöglichkeiten notwendig.

Ansätze für eine derartige Beschreibung können Spezifikations Sprachen bieten. Sie erlauben es anzugeben, wie sich ein System bei bestimmten Eingabeparametern verhalten soll. Sie beschreiben nur den Effekt, den eine Berechnung haben, nicht jedoch wie dieser erreicht werden soll. Daraus läßt sich möglicherweise die Information gewinnen, welche verschiedenen Verhalten als korrekt eingestuft werden sollen. Wenn eine derartige Sprache um Attribute zur Priorität bestimmter Aktivitäten erweitert wird, könnte eine verbesserte Klassifikation möglich sein. Dann wäre es denkbar, das Verhalten eines Systems unter Fehlern nicht nur als „korrekt“ bzw. „falsch“ einzustufen, sondern feiner abgestuft zu beurteilen. Dies würde Aussagen wie z.B. „nach einem Fehler arbeitet das System mit 90% Wahrscheinlichkeit alle als ‘wichtig’ deklarierten Aufgaben korrekt ab“ ermöglichen. Damit wäre eine deutlich verbesserte Klassifikation der Ergebnisse gegeben.



---

# Literatur

- [Aho89] A. V. Aho, R. Sethi, J. D. Ullmann, „Compilers: Principles, Techniques and Tools“, Addison-Wesley, ISBN 0-201-10194-7, 1989.
- [Allmaier97] S. Allmaier, S. Dalibor, "PANDA - Petri Net Analysis and Design Assistant", in Tools Descriptions of the 9th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, St. Malo, Frankreich, Juni 1997, S. 58.
- [Arlat90] 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, Februar 1990, Vol. 16, No. 2, S. 166-182.
- [Armstrong92] J. R. Armstrong, F.-S. Lam, P. C. Ward, "Test Generation and Fault Simulation for Behavioral Models", in "Performance and Fault Modelling with VHDL", (J. M. Schoen, Ed.), Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1992, S. 240-303.
- [Ashenden90] P. J. Ashenden, "The VHDL Cookbook", Technical Report, University of Adelaide, Adelaide, Australia, 1990.
- [Avresky93] D. R. Avresky, J. A. Clark, D. K. Pradhan, "FITS and REACT-tools for Dependability Validation of Fault Tolerant Computing Systems", International Workshop on Fault and Error Injection for Dependability Validation of Computer Systems, Göteborg, Schweden, 17.-18. Juni, 1993.
- [Barton90] J. Barton, E. Czeck, Z. Segall, D. Siewiorek, "Fault Injection Experiments using FIAT", IEEE Transaction on Computers, April 1990, Vol. 39, No. 4, S. 575-582.
- [Benyo93] B. Benyó, "Fault Injection-Based Dependability analysis", Diplomarbeit, Universität Budapest, Budapest, Ungarn, 1993.
- [Bleck96] A. Bleck, "Praktikum des modernen VLSI-Entwurfs", Teubner-Verlag, Stuttgart, ISBN 3-519-02296-6, 1996.
- [Bogendörfer96] R. Bogendörfer, "VHDL-basierte Fehleranalyse von Netzwerkelementen", Diplomarbeit IMMD3, Erlangen, 1996.
- [Böhm94] A. Böhm, "Algorithmenbasierte Fehlertoleranz auf Multiprozessoren: Fehlererkennende Algorithmen in spezialisierter Laufzeitumgebung", Dissertation, Universität Erlangen-Nürnberg, Erlangen, 1994.
- [Bronstein87] I. N. Bronstein, K. A. Semendjajev: "Taschenbuch der Mathematik", Verlag Harri Deutsch, Thun, 1987.

## Literatur

---

- [Buchholz94] P. Buchholz, J. Dunkel, B. Müller-Clostermann, M. Sczittnick, S. Zäske, "Quantitative Systemanalyse mit Markovschen Ketten", B. G. Teubner Verlagsgesellschaft, Stuttgart/Leipzig, 1994.
- [Carreira95a] J. Carreira, H. Madeira, J. G. Silva. "Assessing the Effects of Communication Faults on Parallel Applications", Proceedings of International Computer and Dependability Symposium (IPDS'95), Erlangen, Deutschland, IEEE Computer Society Press, April 1995, S. 214-223.
- [Carreira95b] J. Carreira, H. Madeira, J. G. Silva. "Xception: Software Fault Injection and Monitoring in Processor Functional Units" Proceedings of Working Conference on Dependable Computing for Critical Applications (DCCA-5), Urbana Champaign, USA, Beckman Institute, 27.-29. September, 1995, S. 135-149.
- [Choi92] G. S. Choi, R. K. Iyer, "FOCUS: An Environment for Fault Sensitivity Analysis", IEEE Transactions on Computers, Dezember 1992, Vol. 41, No. 12, S. 1515-1526.
- [Clark93] J. A. Clark, D. K. Pradhan, "REACT: A Synthesis and Evaluation Tool for Fault-Tolerant Multiprocessor Architectures", Proceedings 1993 Annual Reliability and Maintainability Symposium, IEEE Press, Piscataway (N.J.), USA, 1993, S. 428-435.
- [Clark95] J. A. Clark, D. K. Pradhan, "Fault Injection — A Method for Validating Computer-System Dependability" in "Computer Innovative technology for computer professionals", IEEE Computer Society, Juni 1995, Vol. 28, No. 6, S. 47-56.
- [Cusick86] J. Cusick, R. Koga, W. Kolasinski, C. King, "SEU vulnerability of the Zilog Z-80 and NSC-800 microprocessors", IEEE Transactions on Nuclear Science, Dezember 1986, Vol. 32, No. 6, S. 4206-4211.
- [DalCin94] M. Dal Cin, W. Hohl, S. Dalibor, T. Eckert, A. Grygier, H. Hessenauer, U. Hildebrand, J. Hönig, F. Hofmann, C.-U. Linster, E. Michel, A. Pataricza, V. Sieh, T. Thiel, S. Turowski, "Architecture and Realization of the Modular Expandable Multiprocessor System MEMSY", Proceedings 1st Conference on Massively Parallel Computing Systems (MPCS'94), Ischia, Italien, Mai 1994, S. 7-15.
- [Deitel90] H. M. Deitel, "An introduction to operating systems", Addison-Wesley Verlag, 1990.
- [Echtle92] K. Echtle, M. Leu, "The EFA Fault Injector for Fault Tolerant Distributed System Testing", Proceedings of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, Amherst (MA), USA, 1992, S. 28-35.

- 
- [Gaisler97] J. Gaisler, "Evaluation of a 32-bit Microprocessor with Build-In Concurrent Error-Detection", Proceedings 27th International Conference on Fault Tolerant Computing Systems (FTCS-27), Seattle (WA), USA, 24.-27. Juni 1997, S. 42-46.
- [Golze96] U. Golze, "VLSI chip design with the hardware description language VERILOG", Springer Verlag, Berlin, Deutschland, 1996.
- [Goswami90] K. K. Goswami, R. K. Iyer, "DEPEND: A Design Environment for Prediction and Evaluation of System Dependability", Proceedings 9th Digital Avionics Systems Conference, Oktober 1990.
- [Gotthardt68] E. Gotthardt, "Einführung in die Ausgleichsrechnung", Herbert Wichmann Verlag, Karlsruhe, 1968.
- [Güthoff95] J. Güthoff, V. Sieh, "Improving the Efficiency of Fault Injection Based Dependability Evaluation", Proceedings 25th Symposium on Fault-Tolerant Computing (FTCS-25), Pasadena (CA), USA, 27.-30. Juni 1995, S. 196-206.
- [Gunneflo89] U. Gunneflo, J. Karlsson, J. Torin: "Evaluation of error detection schemes using fault injection by heavy-ion radiation.", Proceedings of the 19th International Symposium on Fault-Tolerant Computing (FTCS-19), Chicago (Ill), USA, 21.-23. Juni 1989, S. 340-347.
- [Gunneflo90] U. Gunneflo, "The Effects of Power Supply Disturbances on the MC6809E Microprocessor", Technical Report 89, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Schweden, 1990.
- [Han93] S. Han, H. A. Rosenberg, K. G. Shin: "DOCTOR: An Integrated Software Fault InjeCTiOn EnviRonment", Technical Report, University of Michigan, Dezember 1993.
- [Hein95] A. Hein, K. K. Goswami, "Combined Performance and Dependability Evaluation with Conjoint Simulation", Proceedings 7th European Simulation Symposium, Erlangen, Deutschland, 26.-28. Oktober, 1995, S. 365-369.
- [Hennion85] B. Hennion, P. Senn, "ELDO: A New Third Generation Circuit Simulator Using the One-Step Relaxation Method", International Symposium on Circuits and Systems, Kyoto, Japan, 1985, S. 1065 - 1068
- [Hoffmann75] R. Hoffmann, "The hardware description and programming language HDL", Interner Bericht 75/4, Technische Universität Berlin, Fachbereich Kybernetik, Berlin, 1975.
- [Hönig94] J. Hönig, "Softwaremethoden zur Rückwärtsfehlerbehebung in Hochleistungsparallelrechnern mit verteiltem Speicher", Dissertation, Universität Erlangen-Nürnberg, Erlangen, 1994.

## Literatur

---

- [Iyer86] R. H. Iyer, D. Rosetti, "A measurement based model for workload-dependance of CPU-errors", IEEE Transactions on Computers, Juni 1986, Vol. 35, S. 511-519.
- [Iyer93] R. K. Iyer, D. Tang, "How Many Fault Injections are Necessary", Proceedings International Workshop on Fault Injection for Dependability Validation of Computer Systems, Chalmers University of Technology, Göteborg, Schweden, Juni 1993.
- [Iyer94] R. K. Iyer, "Experimental Analysis of Computer System Dependability", Fault-Tolerant Computing, Second Edition, D. K. Pradhan, Ed., Prentice Hall, 1994.
- [Jacomet91] M. Jacomet, "Layoutabhängige Fehleranalyse und Testsynthese integrierter CMOS Schaltungen", Series in Microelectronics, Vol. 8, Hartung-Gorre Verlag, Konstanz, 1991.
- [Jenn94] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, J. Karlsson: "Fault Injection into VHDL Models: The MEFISTO Tool", Proceedings 24th International Symposium on Fault Tolerant Computing (FTCS-24), IEEE, Austin (Texas), USA, S. 66-75, 1994.
- [Kanawati92] G. Kanawati, N. Kanawati, J. Abraham, "FERRARI: A Tool for the Validation of System Dependability Properties", Proceedings 22th Symposium on Fault-Tolerant Computing (FTCS-22), Boston (MA), USA, 8.-10. Juli 1992, S. 336-344.
- [Kanawati93] G. A. Kanawati, N. A. Kanawati, J. A. Abraham, "EMAX: An Automatic Extractor of High-Level Error Models", Proceedings American Institute of Aeronautics and Astonautics, 1993, S. 1297-1306.
- [Kanawati94] G. A. Kanawati, N. A. Kanawati, J. A. Abraham, "EMAX: An Automatic Extractor of High-Level Error Models", Technical Report, University of Texas at Austin, Computer Engineering Research Center, April 1994.
- [Kao93] W. Kao, R. K. Iyer, D. Tang, "FINE: A Fault Injection and Monitor Environment for Tracing the UNIX System Behavior under Faults", IEEE Transactions on Software Engineering, November 1993, Vol. 19, No. 11, S. 1105-1118.
- [Kao94] W. Kao, R. K. Iyer. "DEFINE: A distributed fault injection and monitoring environment", Proceedings of IEEE Workshop on Fault-tolerant Parallel and Distributed Systems, Juni 1994.
- [Karlsson89] 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 "Dependable Computing and Fault-Tolerant Systems", Springer-Verlag Wien-New York, 1991, Vol. 4, S. 197-212.

- 
- [Kernighan90] B. W. Kernighan, D. M. Ritchie, "Programmieren in C", 2. Auflage, ANSI-C, München, Hanser Verlag, ISBN 3-446-15497-3, 1990.
- [Khare95] J. Khare, W. Maly, "Inductive Contamination Analysis (ICA) with SRAM Application", Proceedings IEEE International Test Conference, Washington (D.C.), USA, Oktober 1995.
- [Kumar95] 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 1st Annual RASSP Conference, Arlington (Virginia), USA, 15.-18. August 1994, S. 114-123.
- [Lala83] J. Lala, "Fault detection, isolation, and reconfiguration in FTMP: Methods and experimental results", Proceedings 5th AIAA/IEEE Digital Avionics Systems Conference (DASC-5), 1983, S. 21.3.1-21.3.9.
- [Lee85] W. S. Lee, D. L. Grosh, F. A. Tillman, C. H. Lie, "Fault Tree Analysis, Methods, and Applications – A Review", IEEE Transactions on Reliability, August 1985, Vol. R-34, No. 3, S. 194-203.
- [Lovric95] Tomislav Lovric, "Processor Fault Simulation with ProFI", Proceedings 7th European Simulation Symposium, Universität Erlangen-Nürnberg, Erlangen, Deutschland, 26.-28. Oktober 1995, S. 353-357.
- [Madeira94] H. Madeira, M. Rela, J. G. Silva. "RIFLE: A General Purpose Pin-Level Fault Injector" Proceedings 1st European Dependable Computing Conference (EDCC-1), Springer Verlag, Berlin, Deutschland, 4.-6. Oktober 1994, S. 199-216.
- [Mahmood88] A. Mahmood., "Concurrent Error Detection Using Watchdog Processors - A Survey", IEEE Transactions on Computers, Februar 1988, Vol. 37, No. 2, S. 160-174.
- [Marsan91] M. A. Marsan, G. Balbo, G. Chiola, G. Conte, S. Donatelli, G. Franceschinis, "An introduction to generalized stochastic Petri-Nets", Microelectronic Reliability, 1991, Vol. 31, No. 4, S. 699-725.
- [Marsan95] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, G. Franceschinis, "Modelling with generalized stochastic Petri-Nets", Wiley, Series in Parallel Computing, 1995.
- [Michel92] E. Michel, "Fehlererkennung mit Überwachungsrechnern in Multiprocessorsystemen", Dissertation, Universität Erlangen-Nürnberg, Erlangen, 1992.
- [Motorola90a] Motorola Inc., "MC88100 Risc Microprocessor User's Manual", Prentice Hall, Englewood Cliffs (NJ), USA, 1990.
- [Motorola90b] Motorola Inc., "MC88200 Cache/Memory Management Unit User's Manual", Prentice Hall, Englewood Cliffs (NJ), USA, 1990.

## Literatur

---

- [Ohlsson92] J. Ohlsson, M. Rimen, U. Gunneflo: "A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog". Proceedings 22nd International Symposium on Fault Tolerant Computing (FTCS-22), Boston (MA), USA, Juli 1992, S. 316-325.
- [Probst94] S. Probst, "Fehlerinjektor für das MACH-Betriebssystem (MEMSY)", Diplomarbeit, IMMD III, Erlangen, November 1994.
- [Riecken95] V. Riecken, "Simulation eines Transputersystems zur Fehlerinjektion", Diplomarbeit, IMMD III, Erlangen, April 1995.
- [Rosenberg93] H. Rosenberg, K. G. Shin. "Software fault injection and its application in distributed systems", Proceedings 23rd International Symposium on Fault-Tolerant Computing (FTCS-23), Frankreich, Juni 1993, S. 208-217.
- [Sahner96] R. A. Sahner, K. S. Trivedi, A. Puliafito, "Performance and reliability analysis of computer systems", Kluwer, Boston (MA), USA, 1996.
- [Saleh87] R. A. Saleh, "Nonlinear Relaxation Algorithms for Circuit Simulation", Memorandum No. UCB/ERL M87/21, Electronics Research Laboratory University of California (Berkeley), USA, 1987.
- [Schwetman86] H. Schwetman, "CSIM: A C-Based Process-Oriented Simulation Language", Proceedings Winter Simulation Conference, 1986.
- [Segall88] 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", Proceedings 18th International Symposium on Fault-Tolerant Computing (FTCS-18), Tokyo, Japan, IEEE Computer Society Press, Juni 1988, S. 102-107.
- [Siewiorek82] D. Siewiorek, R. Swarz, "The Theory and Practice of Reliable Systems Design", Digital Equipment Corporation, 1982.
- [Sieh93] V. Sieh, "Fault-Injector using UNIX-pttrace Interface", Interner Bericht 11/93, Universität Erlangen-Nürnberg, IMMD3, 1993.
- [Sieh94] V. Sieh, A. Pataricza, B. Sallay, W. Hohl, J. Hönig, B. Benyo, "Fault Injection Based Validation of Fault-Tolerant Multiprocessors", Proceedings 8th Symposium on Microcomputer and Microprocessor Applications, Technische Universität Budapest, Ungarn, 12.-14. Oktober 1994, Vol. 1, S. 85-94.
- [Sieh96] V. Sieh, O. Tschäche, F. Balbach, "VHDL-based Fault Injection with VERIFY", Interner Bericht 5/96, Universität Erlangen-Nürnberg, IMMD3, Erlangen, Mai 1996

- 
- [Sieh97a] 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.
- [Sieh97b] V. Sieh, F. Balbach, O. Tschäche, "VERIFY: Zuverlässigkeitsanalyse unter Verwendung von VHDL-Modellen mit integrierter Fehlerbeschreibung", Proceedings 9th Workshop „Testmethoden und Zuverlässigkeit von Schaltungen und Systemen“, Bremen, Deutschland, 9.-11. März 1997, S. 39-42.
- [Sieh97c] V. Sieh, O. Tschäche, F. Balbach, "System Dependability Analysis using VHDL Models with Integrated Fault Descriptions", Extended Abstracts 8th European Workshop Dependable Computing (EWDC-8), Göteborg (Schweden), 1.-4. April 1997.
- [Sieh97d] 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, 23.-27. Juni 1997, S. 32-36.
- [Steininger97] 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, 23.-27. Juni 1997, S. 238-247.
- [Stewart94] W. J. Stewart, "Introduction to the numerical solution of Markov chains", Princeton University Press, Princeton (NJ), USA, 1994.
- [Stiborsky97] J. Stiborsky, "Modellierung eines STC104 in VHDL", Studienarbeit, Universität Erlangen-Nürnberg, IMMD III, Erlangen, August 1997.
- [Tapadiya94] P. K. Tapadiya, D. R. Avresky, "Error Modeling using SOFIT - A Software Fault Injection Tool", Proceedings 3rd International Workshop on Integrating Error Models with Fault injection, Annapolis (MA), USA, April 1994.
- [Thomas91] D. E. Thomas, P. R. Moorby, "The Verilog hardware description language", Kluwer Academic Press, Boston, USA, 1991.
- [Trivedi82] Probability and statistics with reliability, queuing, and computer science applications", Prentice-Hall, Englewood Cliffs (NJ), USA, 1982.
- [Trivedi87] K. S. Trivedi, et. al., "Analysis of Typical Fault-Tolerant Architectures using HARP", in IEEE Transactions on Reliability", Juni 1987, Vol. 36, No. 2.

## Literatur

---

- [Tsai95] T. K. Tsai, R. K. Iyer, "Measuring fault tolerance with the ftape fault injection tool", Proceedings Performance Tools '95 (MMB '95), September 1995, S. 16-40.
- [Tschäche95] O. Tschäche, "Erarbeitung eines detaillierten Fehlermodells für die MEMSY-Hardware", Studienarbeit, Universität Erlangen-Nürnberg, IMMD III, Erlangen, März 1995.
- [Tschäche96a] O. Tschäche, "Automatische Optimierung von VHDL-Modellen", Diplomarbeit, Universität Erlangen-Nürnberg, IMMD III, Erlangen, Februar 1996.
- [Tschäche96b] O. Tschäche, V. Sieh, "ATOMS - A tool for Automatic Optimization of Gate-Level VHDL Models for Simulation", Proceedings 8th European Simulation Symposium (ESS-8), Genua, Italien, 24.-26. Oktober 1996, Volume II, S.329 ff.
- [Yang92a] F. L. Yang, "Simulation of Faults Causing Analog Behavior in Digital Circuits", Dissertation, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Mai 1992.
- [Yang92b] F. L. Yang, R. A. Saleh, "Simulation and Analysis of Transient Faults in Digital Circuits", IEEE Journal of solid-state circuits, März 1992, Vol. 27, No. 3, S. 258-264.
- [Young92a] L. T. Young, R. K. Iyer, "A Hybrid Monitor Assisted Fault Injection Environment", Technischer Bericht, CRHC-92-04, Coordinated Science Laboratory, Urbana Illinois, März 1992.
- [Young92b] L. T. Young, R. K. Iyer, K. K. Goswami, C. Alonso, "A Hybrid Monitor Assisted Fault Injection Environment" in C. E. Landwehr, B. Randell, L. Simoncini (eds.), "Dependable Computing for Critical Applications 3" in "Dependable Computing and Fault-Tolerant Systems", Vol. 8, Springer-Verlag Wien-New York, 1993, S. 281-302.
- [IEEE88] "IEEE Standard VHDL Language Reference Manual", IEEE Std 1076-1987, IEEE Inc., New York (NY), USA, März 1988.
- [IEEE93] "IEEE Standard VHDL Language Reference Manual", ANSI/IEEE Std 1076-1993, IEEE Inc., New York (NY), USA, 1993.
- [USENIX92] Proceedings of the USENIX MACH II Symposium, USENIX Association Berkeley, Monterey (CA), USA, 1992
- [USENIX93] Proceedings of the USENIX MACH III Symposium, USENIX Association Berkeley, Monterey (CA), USA, 1993

# Indexregister

## A

ADEPT 6, 24  
ALPHA-Prozessor 79  
Analysewerkzeug 5  
Anweisung, IF- 17, 48  
Anweisung, WHILE- 17  
Ausfallrate 9, 28, 29, 30, 32  
Ausfallwahrscheinlichkeit 29, 73  
Ausgabe 73  
Ausnahme, Illegal Instruction 26  
Ausnahmebehandlung 68, 69  
äußere Einflüsse 2  
Auswertung 65, 67, 79, 81  
Auswertung, analytische 23  
Auswertung, maschinelle 5  
Auswertung, simulative 23

## B

Benchmark-Abhängigkeit 57, 58  
Beobachtbarkeit 7, 8, 27, 42, 65, 67, 68, 69  
Beschleunigung 37  
Binomialverteilung 30  
Busfehler 11

## C

C (Programmiersprache) 80  
Cache 34, 51  
Clock-Zyklus 8  
Common Cause Fault 27  
Compiler-Bau 48  
Copy-on-Write 37  
CSFI 8, 68  
CSIM 6, 24

## D

Demand Paging 35  
DEPEND 6, 9, 11, 24, 25  
Design-Fehler 80  
Design-Phase 7, 10, 11, 13  
Dhrystone-Benchmark 52, 54, 55, 56, 58  
DOCTOR 8, 68  
Dokumentation 10  
DP32 83, 84

## E

EFA 8, 68  
Einzelfehlerannahme 27, 36  
Einzelschrittanalysator 52, 53  
Einzelschrittmodus 26, 51, 58, 59  
EMAX 7  
Embedded Controller 27

Ereignis 24, 25, 77

Exponentialverteilung 9, 17, 18, 28, 85

## F

fail-safe 1, 2

Farming 38

Fehler, Auftrittsdauer 5, 7, 8, 9, 17, 18, 24, 81

Fehler, Auftrittswahrscheinlichkeit 7, 28

Fehler, Auftrittszeitpunkt 5, 8, 18, 24, 71, 81

Fehler, Bit-Flip- 8, 10, 11, 33, 35, 41, 54, 56, 58, 59, 85

Fehler, Mehrfach- 8

Fehler, Nachrichtenverdopplung 11

Fehler, Nachrichtenverfälschung 8, 11

Fehler, Nachrichtenverlust 8, 11, 35

Fehler, permanenter 26, 27, 34

Fehler, physikalischer 16

Fehler, Pin-Level- 10, 11, 28, 41, 86

Fehler, Register- 50, 56

Fehler, Speicher- 50

Fehler, Stuck-At- 8, 11, 19, 28, 39, 41, 60, 85

Fehler, Stuck-At-Else 7

Fehler, Stuck-At-If 7

Fehler, Stuck-Open- 11

Fehler, Stuck-Toggle- 11

Fehler, temporärer 9, 26, 27, 34, 70

Fehler, Übertragungs- 8

Fehleraktivierung 24

Fehlerausbreitung 60

Fehlerauswirkung 19, 24, 28, 49, 67, 68

Fehlerinjektion 9, 10, 23

Fehlerinjektion, Hardware-basierte 7

Fehlerinjektion, Multi-Threaded 36, 37

Fehlerinjektion, Pin-Level- 26

Fehlerinjektion, Simulationsbasierte 51

Fehlerinjektion, Software-implementierte 26, 35, 51

Fehlerinjektor 8, 9, 10

Fehlerinjektor, Hardware-basiert 10, 26

Fehlerinjektor, Software-basiert 10

Fehlerklasse 33, 67

Fehlerklassifizierung 67

Fehlermaskierung 51, 59, 60, 63

Fehlermenge 9

Fehlermodell 5, 6, 7, 8, 9, 10, 11

Fehlermodell, einfaches 5, 7, 9

## Indexregister

- Fehlermodell, integriertes 6
- Fehlermodell, vollständiges 7, 8
- Fehlerort 8, 26, 81
- Fehlerrate 7, 8, 9, 10, 18, 23, 32, 33
- Fehlersignal 20, 44
- Fehlersimulator 7
- Fehlertoleranzeigenschaft 5
- Fehlertyp 81
- Fehlerüberdeckung 5
- Fehlerursache 27, 28
- FERRARI 8, 34
- FIAT 8, 34, 68
- FINE 8, 68
- FOCUS 7, 25, 68
- FTMP 7, 68
- G**
- Gatterebene 5, 6, 11, 13, 15, 35, 43, 44, 45, 79, 81, 84
- Gattermodell 49
- Genauigkeit, absolute 75
- Genauigkeit, relative 76
- Genauigkeit, statistische 23
- Generic-Parameter 18
- Golden-Run 28, 36, 37, 38, 39, 40, 59, 62, 67, 69, 70, 71, 72, 74, 81
- H**
- Halbleiterfunktion, fehlerhafte 11
- Handshake 69
- Hardware-Instrumentierung 7
- Häufigkeit, relative 28, 30
- HYBRID 7, 68
- I**
- In-Circuit-Emulator 34
- Instrumentierung 6, 7, 8, 26
- Instrumentierung, Software- 7
- K**
- kombinatorischer Block 44, 46, 62
- Konfidenz 23, 24, 29, 33, 37
- Konsistenz 7, 8
- Konvertierungsroutine 42
- Korrespondenz 12, 14
- Kurzschluß 11, 27, 28
- Kurzschluß, Current-Large-Scope 11, 27
- Kurzschluß, Logic-Large-Scope 11, 28
- L**
- Ladung, fehlerhafte 11
- Lastverteilung 38
- Latch 44, 47, 52
- Layout-Ebene 5, 11, 13, 15
- Lebenszyklus 50
- Leistung 5, 11, 37, 42, 63
- Leiterbahn 7
- Leistungsübersprechen 11
- Lesezugriff 49, 50, 51, 53, 59
- M**
- M68020 79
- M88100 52, 53, 79
- M88200 52
- MACH 75
- Material, fehlendes 11
- Material, verändertes 11
- Material, zusätzliches 11
- MEFISTO 7, 10, 24, 25
- Mehrgitterverfahren 55, 56
- Memory-Mapped-IO 69, 74
- MEMSY 52, 79
- MESSALINE 7, 26, 68
- Methodenaufruf 6
- MMU 74
- Model Flattening 45
- Modell 5
- Modell „Advanced“ 85
- Modell „Simple“ 85
- Modell, detailreiches 6
- Modell, strukturelles 5
- Modellerstellung 5, 79
- Modellierung, stochastische 13
- Modellierungssprache 5, 6, 17
- Modellierungswerkzeug 10, 79
- Multiprozessorsystem 7, 26
- Mutant, dynamischer 25
- Mutant, statischer 25
- MVME188 52
- N**
- NOT-Gatter 19, 20
- Nutzungsgrad 54
- Nutzungsgrad, Speicher- 53, 54, 57, 58, 59
- O**
- Operationen, fehlerhafte 11
- Overhead 38, 62
- P**
- Parametrierbarkeit 18
- Pegeländerung 8
- Petri-Netz 9, 11
- Phase, Beobachtungs- 23, 26
- Phase, fehlerlose 23, 26, 33, 34
- Pipeline-Register 8, 34
- Poisson-Differentialgleichung 55
- ProFI 8, 34, 68

- Prototyp 7, 36
- Prozeß 17, 44, 45, 48
- Prozeßsteuerung 1
- Prozeßwechsel 58
- ptrace 26, 58
- R**
- REACT 6, 11, 25
- Read on Demand 35
- Redundanz 2, 59
- Register 44, 47, 49, 51, 52
- Register-Transfer-Ebene 5, 7, 8, 9, 11, 13
- Relaxation 25
- Reparaturkomponente 9
- Ressourcenverbrauch 9, 27
- RIFLE 7, 26, 68
- S**
- Schaltnetz 44
- Schaltungsebene 5, 11, 13, 35
- Schreibzugriff 49, 50, 51, 53, 59
- Schwerionen 8, 10, 26
- SCRIBO 7, 8, 68
- selbstüberwachender Code 55
- Semantik 17
- Sensitivitätsanalyse 63
- Sensitivsignal 49
- SFI 8, 68
- Signal 6, 17, 25, 44, 45
- Signal, analoges 25
- Signal, internes 8
- Signal-Deklaration 18, 19
- Signatur 39, 40, 41
- SimPar 6, 9, 11, 24, 25
- Simulation, digitale 25
- Simulation, Ereignis-gesteuerte 24, 25
- Simulation, hybride 25
- Simulation, kontinuierliche 25
- Simulationsverfahren 23
- SPARC-Prozessor 79
- Speicherschutzmechanismus 27
- Speicherzelle 49, 51
- Speicherzugriff 27
- SPICE 25
- Sprachkonstrukte 17
- Spur 65, 67
- Standardkomponente 32, 33
- Steuerbarkeit 8
- Steuerdatei 7, 8, 9, 11
- Stimulus 49
- Strahlung 27
- Stromversorgung 8, 10, 20, 27
- Synthese 12, 13, 14, 15, 33, 45, 79, 84
- System, deterministisches 24, 39, 65, 70
- System, Echtzeit- 26
- System, Hardware-instrumentiertes 7, 26, 34
- System, instrumentiertes 6, 7, 23, 24, 26, 67, 68
- System, Software-instrumentiertes 8, 26, 27, 34, 68, 69, 74
- System, virtuelles 6, 23, 67
- Systemebene 5, 6, 7, 8, 9
- Systemempfindlichkeit 57
- Systemmodell 5, 6, 7, 9, 10
- Systemverhalten 50
- T**
- Taktrate 10, 26
- Taktzyklus 54, 62, 77
- Teilzustand 67
- Timer-Interrupt 9
- Tri-State-Treiber 52
- U**
- Übertragungsmedium 38
- Umgebungseinflüsse 27
- Unabhängigkeit, statistische 28
- UNIX 26, 27, 58, 79
- Unterbrechung 11
- V**
- Variable 17, 44, 45, 50
- Verhalten, Zeit- 69
- Verhaltensebene 11, 13, 35, 79
- Verhaltensmodell 5, 14, 15, 43
- VERIFY 79, 83, 87
- VERIFY-Auswerter 88
- VERIFY-Compiler 87
- VERIFY-Simulator 88
- VERILOG 6, 17
- Verteilung, Gleich- 28
- VHDL 6, 7, 11, 13, 17, 18, 19, 35, 44, 46, 79, 81
- W**
- Wärme 27
- X**
- XCEPTION 34, 68
- Xception 8
- XOR-Gatter 44, 45
- Z**
- Zähler 44, 47
- Zeitverschiebung 69, 70, 71, 72
- Zellbibliothek 14, 15, 21, 80, 84, 85, 86
- Zugriffsfehler 68

## **Indexregister**

Zustand 44  
Zustand, fehlerhafter 26  
Zustand, interner 5, 49  
Zustand, System- 34, 36, 37, 39, 67, 71,  
73, 74  
Zustand, Teil- 40, 50  
Zustandsänderung 73  
Zustandsübergang 65, 66  
Zustandsvergleich 39, 40, 72, 73, 74, 75  
Zustandsvergleich, inkrementeller 41  
Zuverlässigkeit 5

# Schriftenverzeichnis

## Veröffentlichungen

M. Dal Cin, W. Hohl, S. Dalibor, T. Eckert, A. Grygier, H. Hessenauer, U. Hildebrand, J. Hönig, F. Hofmann, C.-U. Linster, E. Michel, A. Pataricza, V. Sieh, T. Thiel, S. Turowski, "Architecture and Realization of the Modular Expandable Multiprocessor System MEMSY", Proceedings 1st Conference on Massively Parallel Computing Systems (MPCS'94), Ischia, Italien, Mai 1994, S. 7-15.

I. Majzik, A. Pataricza, M. Dal Cin, W. Hohl, J. Hönig, V. Sieh, "Hierarchical Checking of Multiprocessors using Watchdog Processors", Proceedings 1st European Dependable Computing Conference (EDCC-1), Springer LNCS 852, Berlin, Deutschland, 4.-6. Oktober 1994, S. 386-403.

I. Majzik, A. Pataricza, M. Dal Cin, W. Hohl, J. Hönig, V. Sieh, "A High-speed Watchdog Processor for Multitasking Systems", Proceedings 8th Symposium on Microcomputer and Microprocessor Applications ( $\mu$ P'94), Technische Universität Budapest, Budapest, Ungarn, 12.-14. Oktober 1994, Band I, S. 65-74.

V. Sieh, A. Pataricza, B. Sallay, W. Hohl, J. Hönig, B. Benyo, "Fault Injection Based Validation of Fault-Tolerant Multiprocessors", Proceedings 8th Symposium on Microcomputer and Microprocessor Applications ( $\mu$ P'94), Technische Universität Budapest, Budapest, Ungarn, 12.-14. Oktober 1994, Band I, S. 85-94.

J. Güthoff, V. Sieh, "Combining Software-Implemented and Simulation-Based Fault Injection into a Single Fault Injection Method", Proceedings 25th Symposium on Fault-Tolerant Computing (FTCS-25), Pasadena (CA), USA, 27.-30. Juni 1995, S. 196-206.

O. Tschäche, V. Sieh, "ATOMS - A tool for Automatic Optimization of Gate-Level VHDL Models for Simulation", Proceedings 8th European Simulation Symposium (ESS-8), Genua, Italien, 24.-26. Oktober 1996, Band II, S.329-333.

I. Majzik, W. Hohl, A. Pataricza, V. Sieh, "Multiprocessor Checking Using Watchdog Processors", Computer Systems Science & Engineering, Vol. 11, No. 5, 1996, S. 301-310.

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.

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.

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.

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.

M. Dal Cin, W. Hohl, V. Sieh, "Hardware-Supported Fault Tolerance for Multiprocessors", Proceedings 14. ITG/GI-Fachtagung Architektur von Rechensystemen (ARCS'97), Rostock, Deutschland, 8.-11. September, 1997, VDE-Verlag, S. 13-22.

### **Interne Berichte**

V. Sieh, "Fault-Injector using UNIX-pttrace Interface", Interner Bericht 11/93, Universität Erlangen-Nürnberg, IMMD3, Erlangen, Dezember 1993.

V. Sieh, J. Hönig, "Software-Based Concurrent Control Flow Checking", Interner Bericht 10/95, Universität Erlangen-Nürnberg, IMMD3, Erlangen, Dezember 1995.

V. Sieh, O. Tschäche, F. Balbach, "VHDL-based Fault Injection with VERIFY", Interner Bericht 5/96, Universität Erlangen-Nürnberg, IMMD3, Erlangen, Mai 1996.

F. Balbach (ed.), M. Dal Cin, A. Hein, J. Meixner, B. Rügenapp, V. Sieh, J. Stiborsky, O. Tschäche, "Simulationsbasierte Zuverlässigkeitssanalyse", Interner Bericht 6/96, Universität Erlangen-Nürnberg, IMMD3, Erlangen, Juli 1996.

# Lebenslauf

## Person:

Name: Volkmar Sieh  
Geburtsdag: 16. August 1965  
Geburtsort: Flensburg  
Familienstand: ledig  
Staatsangehörigkeit: deutsch

## Schulbesuch:

Juli '72 - Juni '76 Besuch der Grundschule Friedheim in Flensburg  
Juli '76 - Juni '85 Besuch des Fördergymnasiums in Flensburg

## Wehrdienst:

Juli '85 - Juni '87 Bundeswehrdienst in Lüneburg und Flensburg-Weiche

## Studium:

Okt. '87 - März '93 Informatikstudium (Nebenfach Elektrotechnik) an der  
Universität Erlangen-Nürnberg in Erlangen.  
Apr. '93 - jetzt wissenschaftlicher Angestellter am IMMD3 an der  
Universität Erlangen-Nürnberg in Erlangen

## Berufliche Tätigkeiten:

Juli '87 - September '87:  
Mitarbeit im electronic-computer-laden OHG in Flensburg  
Kundenberatung, Verkauf u. Reparatur von Home- und Personal-Computern

August '90, '91, '92 (während des Studiums in den Semesterferien):  
Hilfskraft an der Universität Erlangen-Nürnberg  
Betreuung von rechnergestützten Programmierübungen

Dezember '91 - April '92 (während des Studiums; 20 Stunden pro Woche):  
Werkstudent bei der Firma IPCAS GmbH in Erlangen  
Kundenberatung, Verkauf und Montage von Sun- und Tatung-Workstations,  
Systemadministration

