Transformationen zur paralleliserten und fehlertoleranten Informationsverarbeitung

Felix C. Gärtner Marc Theisen

Graduiertenkolleg ISIA
Technische Universität Darmstadt
Karlstraße 15, 64283 Darmstadt
felix@informatik.tu-darmstadt.de
theisen@mes.tu-darmstadt.de

Zusammenfassung Das durch die Deutsche Forschungsgemeinschaft (DFG) geförderte Graduiertenkolleg "Intelligente Systeme für die Informations- und Automatisierungstechnik" (ISIA) untersucht und erarbeitet Methoden und Mechanismen zur verbesserten Entwicklung von intelligenten Systemen. In diesem Artikel werden beispielhaft zwei im Rahmen dieses Projektes entstandene Arbeiten vorgestellt, die ein gemeinsames Grundkonzept in zwei Varianten erforschen. Beide Arbeiten benutzen Eigenschaftstransformationen, zum einen zur Steigerung der Effizienz, zum anderern zur Verbesserung der Zuverlässigkeit/Fehlertoleranz von informationsverarbeitenden Systemen.

1 Einführung

1.1 Das Graduiertenkolleg

Das Graduiertenkolleg "Intelligente Systeme für die Informations- und Automatisierungstechnik" (ISIA, Sprecher Prof. Dr. Dr. h. c. mult. M. Glesner) an der Technischen Universität Darmstadt hat zum Ziel, Forschungsaktivitäten auf dem Gebiet der intelligenten Systeme zu verstärken. Dieses Projekt läuft seit mehr als sieben Jahren, und es sind 14 Institute aus den Fachbereichen "Elektrotechnik und Informationstechnik" und "Informatik" beteiligt. Insgesamt arbeiten an diesem Forschungsprojekt momentan 13 Stipendiaten und vier Kollegiaten, die damit die Gelegenheit bekommen, ihre Promotion innerhalb von drei Jahren durchzuführen.

In diesem Graduiertenkolleg werden unterschiedliche Aspekte von intelligenten Systemen in den Bereichen der Informations- und Automatisierungstechnik untersucht. Im Rahmen der Kommunikationstechnik werden beispielsweise Verfahren zur Geräuschreduktion mit Neuro- bzw. Fuzzy-Systemen oder Schedulingstrategien für Datenströme in multimedialen Systemen untersucht. Kernpunkte der Arbeiten auf dem Gebiet der Regelungs- und Automatisierungstechnik sowie Robotik sind intelligente Regelsysteme und die Fehlerdiagnose mit neuronalen Netzen. Die in diesen Arbeiten entworfenen Konzepte müssen in einem informationsverarbeitenden System integriert und umgesetzt werden. Hierbei spielen unter anderem neue Hochfrequenztechniken, sowie auf dem Gebiet der Rechnersysteme und der mikroelektronischen Systeme Verfahren zur Optimierung, Synthese und Verifikation von hochintegrierten mikroelektronischen

Schaltungen eine Rolle. Ein zentraler Gedanke in diesem Graduiertenkolleg ist es, das Zusammenspiel von Hardware und Software genauer zu beleuchten. Daher laufen intensive Arbeiten auf den Gebieten der Betriebssyteme und der Mikroelektronik inklusive Halbleitertechnologie.

1.2 Das gemeinsame Grundkonzept der hier vorgestellten Arbeiten

Allen Arbeiten gemeinsam ist der einheitliche Systemgedanke. Zentrale Anforderungen an die zu entwickelnden Systeme sind, daß die Schaltungen möglichst klein und verlustleistungsarm entworfen werden. Dies ist insbesondere für die mobile Kommunikationstechnik vonnöten. In der Sprach- und Bildverarbeitung ist außerdem ein möglichst hoher Datendurchsatz wichtig. Von hoher Bedeutung ist speziell in der Flugsicherheit, daß sich die Systeme fehlertolerant verhalten. Aufgrund der hohen Komplexität der zu entwerfenden Systeme ist es dem Designer nur mit einem großen Arbeitsaufwand möglich, diese Ziele schnell und effizient in einem System zu konkretisieren. Daher sollen im folgenden exemplarisch für die Arbeiten im Graduiertenkolleg Methoden vorgestellt werden, die es erlauben, daß die Systeme interaktiv oder automatisch bezüglich dieser Kriterien optimiert werden. Dabei wird davon ausgegangen, daß eine Spezifikation (Beschreibung des Systemverhaltens) gegeben ist. Bei der Realisierung muß die Beschreibung dann so verändert werden, daß die Optimierungskriterien erfüllt werden (siehe Abbildung 1). In diesem Prozeß ist der Begriff der Transformation von zentraler Wichtigkeit. Darunter soll im folgenden eine Abbildung $f : S \to S$ verstanden werden, wobei S eine Beschreibungsdomäne ist. Urbild- und Bildbereich sind also bei den hier betrachteten Transformationen gleich, und es handelt sich jeweils um eine Menge von Systemspezifikationen, beispielsweise einer Verhaltensbeschreibung einer mikroelektronischen Schaltung, eines Programms, etc.

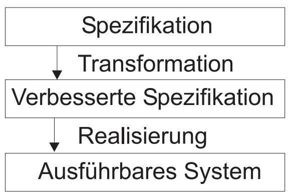

Abbildung 1. Optimierung und Realisierung eines Systems

2 Parallelisierende Transformationen zur Effizienzsteigerung

Schleifen stellen in einem Algorithmus den Teil dar, der am rechenintensivsten ist. Daher wirken sich bei ihnen Transformationen zur Effizienzsteigerung besonders günstig aus. Das Ziel der im folgenden betrachteten Schleifentransformationen [6] soll es daher sein, ein sogenanntes Schleifennest auf seinen optimalen Parallelitätsgrad zu bringen.

Unter einem Schleifennest versteht man ineinander geschachtelte Schleifen. Es muß also der optimale Kompromiß zwischen der benötigten Ausführungszeit und dem Hardwareaufwand gefunden werden. Dieser ist durch die Anzahl der parallel prozessierenden Elemente gegeben [15]. Dazu geht man davon aus, daß das Schleifennest in einer Hardwarebeschreibungssprache (hier VHDL) spezifiziert ist (als Beispiel siehe Abbildung 2 oben links). Diese Spezifikation wird syntaktisch analysiert und der Indexraum und die Indexfunktionen aus dem Schleifennest extrahiert. Daraus ergeben sich die Abhängigkeiten zwischen den Feldzugriffen, also die Abhängigkeitsvektoren und die Abhängigkeitsmatrix. Im nächsten Schritt wird das nur sequentiell ausführbare Schleifennest so transformiert, daß einige Instruktionssequenzen parallel ausgeführt werden können. Dies kann durch eine unimodulare Transformation [4, 5, 17] erreicht werden, die das Schleifennest so transformiert, daß es mit dem maximalen Parallelitätsgrad ausgeführt werden kann. Aber eine Hardwarerealisierung, die mit dem maximalen Parallelitätsgrad arbeitet, ist häufig nicht die effizienteste Ausführungsmöglichkeit, d.h. in vielen Zeitschritten ruht ein hoher Anteil der prozessierenden Elemente und nur in einigen wenigen rechnen alle Elemente gleichzeitig. Um die Effizienz zu steigern, wird daher der Parallelitätsgrad reduziert. Dazu wird der Indexraum einer Koordinatentransformation unterzogen. Nun kann die Relation berechnet werden, die den Zusammenhang zwischen der Anzahl der zur vollständigen Ausführung benötigten Zeitschritte und dem Parallelitätsgrad, also der Anzahl der prozessierenden Elemente, darstellt. Schließlich kann aufgrund der aufgestellten Relation und einer Kostenfunktion, die von der Anwendung der Schaltung und von der zugrundeliegenden Hardwarearchitektur abhängt, der optimale Kompromiß zwischen Fläche und Zeit ermittelt wird. Im nächsten Schritt muß diese parallelisierte Verhaltensbeschreibung zu einer Schaltung synthetisiert werden, d.h. es müssen die Operationen den Ausführungselementen und den Zeitschritten zugeordnet werden. In dem in Abbildung 2 (Mitte) dargestellten Beispiel repräsentiert die äußere sequentielle Schleife die Zuordung zu den einzelnen Zeitschritten und die innere Schleife die Zuordnung zu den prozessierenden Elementen. Durch dieses Verfahren ist also eine optimierte Synthese gelungen.

3 Transformationen zur Verbesserung der Fehlertoleranz

Effizienz ist nur eine von vielen möglichen Eigenschaften, die man durch Transformationen verbessern kann. Andere Eigenschaften, die in der Praxis immer wichtiger werden, sind Zuverlässigkeit, Verfügbarkeit und Sicherheit von technischen Systemen. Die Steuerungssysteme eines Verkehrsflugzeuges müssen beispielsweise so entworfen sein, daß die Wahrscheinlichkeit eines Totalausfalles bei einem einstündigen Flug in der Größenordnung von 10^{-9} liegt [14]. Zur Erhöhung der Zuverlässigkeit kann man natürlich ausschließlich zuverlässige Komponenten verwenden. Jedoch unterliegt jedes physikalische System ab einem gewissen Zeitpunkt einem Ermüdungsprozeß; Fehler und Fehlverhalten einzelner Komponenten sind die Folge. Systeme, die trotz interner Fehler immer noch ein gewünschtes Systemverhalten aufweisen, nennt man *fehlertolerant.*

Fehlertoleranz basiert immer auf irgendeiner Form von Redundanz. Eine Form von Redundanz besteht darin, kritische Komponenten mehrfach in das System einzubringen. Beim Ausfall einer Komponente kann eine andere Komponente (sofort) die Aufga-

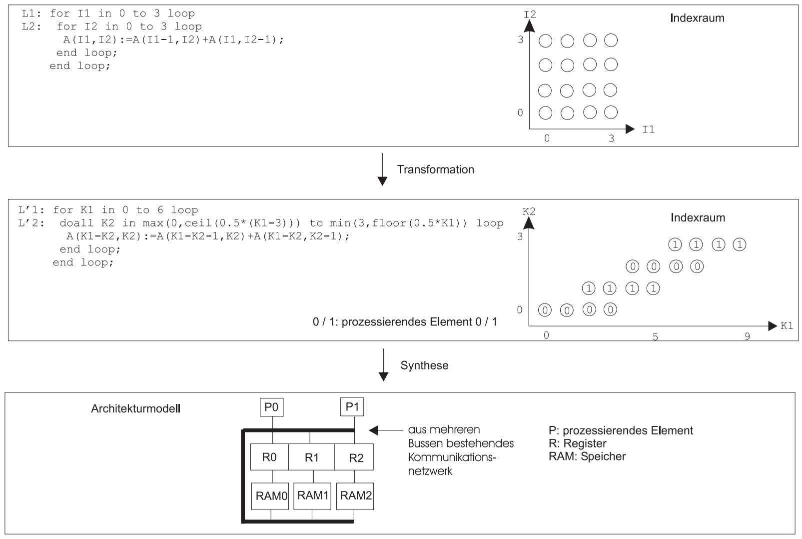

Abbildung 2. Strukturelle Schleifenparallelisierung und Architekturmodell

ben der ausgefallenen Komponente übernehmen [16]. Die Idee ist nun, Redundanz und die dazugehörigen Verwaltungsmechanismen in mehr oder weniger mechanischer Form durch eine Transformation R in ein System zu integrieren. Diese Transformation ist dergestalt, daß sie, gegeben ein System A, für das transformierte System $A' = R(A)$ bestimmte Fehlertoleranzeigenschaften herstellt und formal nachweisbar garantiert. Dies vereinfacht die notwenige Validierung des Systems: Bestimmte Fehlertoleranzmechanismen werden ein für alle mal durch eine Transformation beschrieben und ihre Eigenschaften verifiziert. Ausgehend von den Eigenschaften von A kennt man nun auch die Eigenschaften von A'. Würde man die Mechanismen wie früher immer im Einzelfall "von Hand" einbauen, müßte man jedes fehlertolerante System A' neu verifizieren; ein mühsames Unterfangen.

3.1 Verifikation, Spezifikation, Verhalten und Korrektheit

Jeder Ansatz zu einer Verifikation benötigt eine möglichst genau formulierte Spezifikation. Für die formale Verifikation basieren derartige Spezifikationen auf einem formalen Systemmodell, welches Systeme als nichtdeterministische Automaten modelliert. Ein solcher Automat ist gegeben durch eine Zustandsmenge Z, eine Teilmenge von Startzuständen $I \subseteq Z$ und eine Zustandsübergangsrelation δ. Ein möglicher zeitlicher Ablauf des Systems entsteht wie folgt: ausgehend von einem Zustand z_0 aus I wird durch Benutzung von δ ein Folgezustand z_1 berechnet. Durch wiederholte Inspektion von δ kann man nun $z_2, z_3, \ldots$ bestimmen. Offensichtlich gibt es jedoch wegen des inhärenten Nichtdeterminismus nicht nur einen möglichen Ablauf sondern viele mögliche Systemabläufe.

Eine formale Spezifikation S (manchmal auch *Eigenschaft* genannt) ist in diesem Modell gegeben als Menge von (gewünschten) Abläufen. In der formalen Verifikation geht es nun darum zu zeigen, daß ein System A dieser Spezifikation entspricht. Dies ist gewährleistet, wenn jede mögliche Ausführung von A auch in S enthalten ist. So wird *Korrektheit* definiert. Wenn A also einen Ablauf zeigen kann, der nicht in S liegt, ist A nicht korrekt bezüglich S. Technisch gesehen "reduziert" sich das Problem der Verifikation auf den Nachweis der Gültigkeit einer logischen Formel in einem formalen System [12].

3.2 Arten von Fehlertoleranz

Fehlertoleranz bedeutet im wesentlichen *Korrektheit unter Fehlereinflüssen*. Um diesen Terminus formal zu fassen, muß man also auch den Begriff "Fehler" formalisieren. Hierzu werden formale Fehlermodelle herangezogen [8, 9]. Ein *Fehlermodell* ist eine präzise Beschreibung, wie sich das Verhalten von Komponenten aufgrund eines Fehlers ändert. Ein oft herangezogenes Beispiel ist das Fehlermodell *Teilausfall* (engl. *crash*). Es besagt, daß von einer gegebenen Menge von n gleichen Komponenten maximal $t < n$ plötzlich und dauerhaft aufhören zu arbeiten. Ein Fehlermodell ist immer systembezogen.

Fehlermodelle verändern offensichtlich die Eigenschaften von Systemen (und können darum interessanterweise auch als Transformationen aufgefaßt werden [10, 13]). Wenn man also ein System A konstruiert hat, welches einer Spezifikation S genügt, dann wird A unter einem gegebenen Fehlermodell F möglicherweise nicht mehr dieselben Eigenschaften haben wie vorher. Das System ist dann möglicherweise nicht mehr korrekt bezüglich der Originalspezifikation S sondern genügt lediglich einer "schwächeren" Spezifikation S'.

Nun gibt es zwei grundlegende Klassen von Systemeigenschaften, die als *Sicherheitseigenschaften* (engl. *safety*) und *Lebendigkeitseigenschaften* (engl. *liveness*) bezeichnet werden [1, 11]. Informal besagt eine Sicherheitseigenschaft, daß eine Menge von erwünschten, guten, sicheren Systemzuständen nie verlassen wird. Auf der anderen Seite beschreiben Lebendigkeitseigenschaften, daß das System nach und nach auch einen Fortschritt zeigt und gewünschte Systemzustände annimmt. Basierend auf dieser Unterteilung kann man drei grundlegende Arten von Fehlertoleranz unterscheiden [3].

Nehmen wir also an, wir haben ein System A, welches der Spezifikation S genügt. Wenn unter dem Fehlermodell F sowohl die Sicherheits- als auch die Lebendigkeitseigenschaften von A erhalten bleiben, dann spricht man von *maskierenden Fehlertoleranz* (formal: A maskiert F bezüglich S). Maskierende Fehlertoleranz ist die stärkste und natürlich auch teuerste Art von Fehlertoleranz, die man erreichen kann. Doch offensichtlich gibt es immer Systemeigenschaften, die bei einem hinreichend "bösartigen" Fehlermodell nicht bewahrt werden können (wenn beispielsweise vorübergehende Störeinflüsse wie kosmische Strahlung auf Raumfahrzeuge nur lange genug anhalten, können bestimmte Echtzeiteigenschaften nicht eingehalten werden). Maskierende Fehlertoleranz kann also manchmal schlichtweg unmöglich sein.

Wenn das Fehlermodell nur die Lebendigkeitseigenschaften beeinflußt und die Sicherheitseigenschaften nicht berührt, dann spricht man von *fail-safe-Fehlertoleranz* (formal: A ist *fail-safe*-fehlertolerant für F bezüglich S). Diese Form von Fehlertoleranz ist immer dann sinnvoll, wenn es eine "*safe alternative to normal operation*"

[14] existiert. In Atomkraftwerken ist man beispielsweise daran interessiert, daß das System in einen sicheren Zustand herunterfährt, wenn bestimmte Fehler auftreten.

Statt den Lebendigkeitseigenschaften könnte das Fehlermodell aber auch die Sicherheitseigenschaften beeinflussen. Es ist bisher noch unklar, ob Systeme, die ausschließlich "*live*" sind (und nicht "*safe*") irgendeine praktische Bedeutung haben können. Darum fordert man bei der dritten Art von Fehlertoleranz, daß die Sicherheitseigenschaften aus S zwar verletzt werden können, das System aber nach endlicher Zeit wieder in einen sicheren Zustand zurückkehrt. In diesem Fall spricht man von *nichtmaskierender Fehlertoleranz* (formal: A ist nichtmaskierend-fehlertolerant für F bezüglich S). Beispiele für nichtmaskierend-fehlertolerante Systeme kommen aus dem Gebiet der Selbststabilisierung [7]. Dort ist gefordert, daß ein System nach dem Auftreten eines *beliebigen* vorübergehenden Fehlers nach endlicher Zeit wieder in einen sicheren Systemzustand übergeht. Praktische Anwendung findet dieses Konzept immer dann, wenn die Möglichkeiten anderer Fehlertoleranzarten aufhören. Beispielsweise werden Flugzeugsteuerungen so entworfen, daß sie eine bestimmte (vorher festgelegte) Anzahl und Art von Fehlern tolerieren. Sollte es der Fall sein, daß kurzzeitig diese Fehlerannahme verletzt wird (zum Beispiel wenn im letzten noch funktionierenden Steuerungsrechner ein Bit "kippt"), so möchte man zumindest die Zeit, in der sich das System in einem unsicheren Zustand befindet "minimieren".

3.3 Fehlererkennung und Fehlerkorrektur

Nun stellt sich natürlich die Frage, wie man bestimmte Arten von Fehlertoleranz durch einen möglichst mechanischen Konstruktionsprozeß für ein beliebiges System mit Spezifikation S erreichen kann? Im Hinblick auf eine Antwort auf diese Frage gibt es ein interessantes theoretisches Resultat: Man kann jede Form von Fehlertoleranz mittels zweier einfacher Komponenten erreichen. Diese Komponenten werden *Detektoren* und *Korrektoren* genannt [2]. Informal gesprochen ist ein Detektor eine Systemkomponente, die einen bestimmten Systemzustand erkennt und meldet. Ein Korrektor hingegen ist eine Komponente, die auf Anweisung einen bestimmten Systemzustand "herstellt". (Detektoren und Korrektoren sind im wesentlichen Formalisierungen von Sensoren und Aktoren, die aus vielen Ingenieurbereichen bekannt sind.) Man kann nun zeigen [3], daß Detektoren notwendig und hinreichend sind, um Sicherheitseigenschaften zu gewährleisten. Auf der anderen Seite sind Korrektoren notwendig und hinreichend für die Beibehaltung von Lebendigkeitseigenschaften.

Angenommen wir haben nun ein System A, welches eine der drei Arten von Fehlertoleranz bezüglich einer Spezifikation S erreichen soll. Der Entwurfsablauf für eine "fehlertolerante Version" von A kann nun in die folgenden drei Schritte gegliedert werden:

1. Festlegung des Fehlermodells F. Das Problem hierbei liegt darin, ein Fehlermodell zu finden, welches einerseits "bösartig" genug ist, um alle im Einsatzbereich erwarteten Fehler abzudecken, und gleichzeitig "gutartig" genug, um mit möglichst wenig zusätzlichen Fehlertoleranzmechanismen auszukommen.
2. Entwurf und Realisierung eines Fehlertoleranzverfahrens. Hierzu werden ausgehend vom gegebenen Fehlermodell entsprechende Detektoren und Korrektoren in das System integriert.

3. Verifikation und Validierung. Hierbei wird überprüft, ob das System unter Fehlereinflüssen aus F tatsächlich die geforderte Art von Fehlertoleranz garantiert. Dies kann wie oben beschrieben auch mit Methoden der formalen Verifikation geschehen.

Die Idee ist nun, für bestimmte vorher festgelegte Systemmodelle die Schritte 2 und 3 automatisiert im Rahmen einer Transformation R durchzuführen. Aus dem originalen System A entsteht ein transformiertes System $A' = R(A)$, welches die geforderten Fehlertoleranzeigenschaften für F bezüglich S hat. Dabei sollen die Eigenschaften von A direkt ableitbar sein aus der Beschaffenheit von R.

3.4 Beispiel einer Fehlertoleranztransformation

Die in Abbildung 2 (oben) gegebene Schleife ist auf das Ziel hin transformiert worden, daß sie einer effizienten Realisierung in einer mikroelektronischen Schaltung entspricht (Abbildung 2, Mitte). Diese beiden Spezifikationen sind unter dem Namen S1 und S2 in Abbildung 3 dargestellt. Die Spezifikation S2 besitzt noch keine Fehlertoleranzeigenschaften. Diese sollen mittels einer zweiten Transformation hinzugefügt werden. S2 besitzt die Systemeigenschaft, bei zwei prozessierenden Elementen zur vollständigen Ausführung zehn Iterationsschritte zu benötigen. Diese Eigenschaft soll so transformiert werden, daß das beschriebene System auch bei Ausfall eines prozessierenden Elementes funktioniert. Es besteht einerseits die Möglichkeit, die Systemeigenschaften durch eine Fehlertoleranztransformation R1 derart zu verändern, daß ein drittes prozessierendes Element hinzugefügt wird (Spezifikation S3). Hier ist zu prüfen, ob die Systemanforderungen an die auf dem Chip zur Verfügung stehende Fläche erfüllt bleiben. Andererseits kann die Spezifikation durch eine alternative Transformation R2 so verändert werden, daß sie auf nur einem prozessierenden Element ausgeführt wird (Spezifikation S3*). In diesem Fall ist insbesondere bei einem Echtzeitsystem sicherzustellen, daß die Zeitanforderungen gewahrt bleiben. Insbesondere der Übergang von S2 nach S3* zeigt, daß es Tranformationen mit sich gegenseitig aufhebenden Zielen gibt.

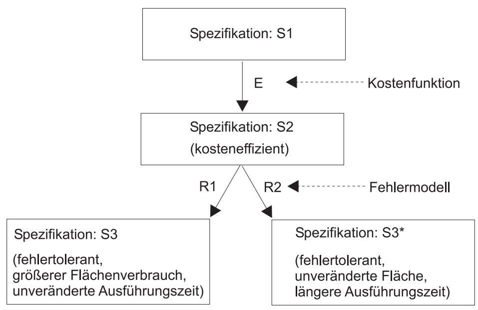

Abbildung 3. Komposition von Transformationen.

4 Ausblick

Wie in Abbildung 3 deutlich wird, können Transformationen auch hintereinander ausgeführt werden. Beispielsweise sei E sei Transformation aus Abschnitt 2 und R die Transformation aus obigem Beispiel, ist dann das System $R(E(S1))$ parallelisiert und fehlertolerant?

Antworten auf diese und ähnliche Fragen sind nicht immer einfach. Grundlage einer Antwort wird immer eine genaue Beschreibung der Wirkung einer Transformation auf die Systemeigenschaften sein. Die Untersuchung solcher Fragestellungen ist Grundlage unserer weiteren interdisziplinären Zusammenarbeit.

Literatur

1. B. Alpern und F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
2. A. Arora und S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *Proc. of the 18th IEEE Int. Conf. on Distr. Comp. Sys.*, Mai 1998.
3. A. Arora und S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, Jan. 1998.
4. U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, 1993.
5. U. Banerjee. *Loop Transformations for Restructuring Compilers: Loop Parallelization.* Kluwer Academic Publishers, 1994.
6. J. Becker. *A Partitioning Compiler for Computers with Xputer-based Accelerators.* Dissertation, Universität Kaiserslautern, 1997.
7. E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
8. K. Echtle. Fehlermodellierung bei Simulation und Verifikation von Fehlertoleranz-Algorithmen für Verteilte Systeme. In F. Belli, S. Pfleger, und M. Seifert (Hg.), *Software-Fehlertoleranz und -Zuverlässigkeit*, Inf.-Fachberichte (Bd. 83), S. 73–88. Springer, 1984.
9. K. Echtle und A. Masum. Understanding cooperative byzantine failures: A novel failure classification to enable efficient fault-tolerant protocols. In *Proc. of the IEEE Workshop on Fault-Tolerant Parallel and Distr. Sys.*, San Juan, Puerto Rico, USA, Apr. 1999. Kluwer.
10. F. C. Gärtner. Specifications for fault tolerance: A comedy of failures. Technischer Bericht TUD-BS-1998-03, TU Darmstadt, Fachbereich Informatik, Okt. 1998.
11. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, März 1977.
12. L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, Jan. 1989.
13. Z. Liu und M. Joseph. Transformation of programs for fault-tolerance. *Formal Aspects of Computing*, 4(5):442–469, 1992.
14. J. Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and System Safety*, 43(2):189–219, 1994.
15. O. Schneider. Transformationen von Schleifenindizes zur Parallelisierung von VHDL-Verhaltenscode. Diplomarbeit, TU Darmstadt, 1999.
16. A. Spector und D. Gifford. The space shuttle primary computer system. *Communications of the ACM*, 27(9):874–900, 1984.
17. M. Theisen, J. Becker, M. Glesner, und T. Caohuu. Parallel hardware compilation in complex hardware/software systems based on high-level code transformations. In *ARCS'99: 15. GI/ITG-Fachtagung: Architektur von Rechensystemen*, Jena, Okt. 1999.