Formale Grundlagen der Fehlertoleranz in verteilten Systemen

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades eines Doktor-Ingenieurs (Dr.-Ing.)
vorgelegt von

Dipl.-Inform. Felix Christoph Gärtner

aus Korbach

Referenten:

Prof. Dr. Peter Kammerer

Prof. Dr. Friedemann Mattern

Datum der Einreichung: 20. März 2001

Datum der mündlichen Prüfung: 17. Mai 2001

Darmstädter Dissertationen D17

Erklärung

Hiermit erkläre ich, die vorgelegte Arbeit zur Erlangung des akademischen Grades "Dr.-Ing." mit dem Titel "Formale Grundlagen der Fehlertoleranz in verteilten Systemen" selbständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel erstellt zu haben. Ich habe bisher noch keinen Promotionsversuch unternommen.

Darmstadt, 11. Juli 2001 Felix Gärtner

Kurzfassung

Fehlertolerante Systeme arbeiten auch dann noch korrekt, wenn während ihrer Ausführung eine bestimmte Anzahl von Fehlern auftreten. Um fehlertolerante Systeme zu entwickeln und zu validieren, benötigt man eine wissenschaftlich gesicherte Methodik. Bestandteile einer solchen Methodik sind Verfahren der Modellbildung, Entwurfsmuster für Fehlertoleranzverfahren und vorgefertigte algorithmische Grundbausteine. Diese Arbeit trägt zu jedem dieser drei Bereiche bei.

Im Rahmen einer allgemeinen Modelluntersuchung werden in Kapitel 2 vier wichtige Systemmodelle für fehlertolerante verteilte Systeme vorgestellt und miteinander verglichen: das (partiell) synchrone Modell von Dwork, Lynch und Stockmeyer, das Fehlerdetektormodell von Chandra und Toueg, das zeitbeschränkte asynchrone Modell von Cristian und Fetzer, sowie das quasi-synchrone Modell von Almeida, Veríssimo und Casimiro.

Kapitel 3 untersucht die grundsätzliche Funktionsweise von Fehlertoleranzverfahren im Rahmen der Fehlertoleranztheorie von Arora und Kulkarni. In dieser Theorie entstehen fehlertolerante Programme aus der Komposition eines fehler-intoleranten Programms mit Fehlertoleranzkomponenten. Kapitel 3 erweitert diese Theorie um eine präzise Begrifflichkeit der *Redundanz*. Es werden zwei grundlegende Formen von Redundanz identifiziert (Speicherredundanz und Zeitredundanz) und es wird gezeigt, welche Art von Redundanz notwendig ist, um auch im Fehlerfall bestimmte Eigenschaften zu erfüllen.

Kapitel 4 beschäftigt sich mit Lösungen des *Beobachtungsproblems* in verteilten Systemen, in denen Fehler auftreten können. Beobachten heißt hierbei, zu wissen, ob ein globales Prädikat während der Ausführung eines Algorithmus gilt oder nicht. Es werden zwei neue Beobachtungsmodalitäten definiert für die Entdeckung allgemeiner Prädikate in asynchronen Systemen, in denen *crash*-Fehler auftreten können. Die neuen Modalitäten können als Erweiterungen der aus den fehlerfreien Systemen bekannten Beobachtungsmodalitäten *possibly* und *definitely* von Cooper, Marzullo und Neiger angesehen werden. Abschließend werden Entdeckungsalgorithmen für die neuen Modalitäten vorgestellt und verifiziert.

Abstract

A system is fault-tolerant if it maintains some form of correctness in the presence of faults. Fault-tolerant systems are necessary in many application areas of computing, especially where the failure of a computer system may lead to considerable damage. However, the development of fault-tolerant systems is a difficult task and requires rigorous and scientifically sound engineering methodologies. Amoung other topics, these methodologies must cover (1) system and fault modelling, (2) design patterns for fault-tolerance mechanisms and (3) basic building blocks for fault-tolerant algorithms. This thesis contributes to research in all of these three areas.

Questions of system and fault modelling are treated in chapter 2 in which the four most prominent system models for fault-tolerant distributed systems are presented and compared: the model of partial synchrony by Dwork, Lynch and Stockmeyer, the failure detector model of Chandra and Toueg, the timed asynchronous system model of Cristian and Fetzer, and the quasi-synchronous model by Almeida, Veríssimo and Casimiro.

Chapter 3 focusses on the underlying principles of fault-tolerant systems. Starting point is the fault-tolerance theory of Arora and Kulkarni. In this theory, a fault-tolerant program is regarded as the composition of a fault-intolerant program and fault-tolerance components. We analyse the assumptions of the theory and extend it by defining formal notions of *redundancy*. Two forms of redundancy are identified (redundancy in space and redundancy in time) and we study which forms of redundancy are necessary to maintain which properties in the presence of faults.

In chapter 4 we develop algorithmical building blocks for *observation* in faulty environments, a central problem in fault-tolerant computing which has not enjoyed a lot of research attention yet. Observation here means detecting whether or not a boolean predicate on global states holds during the execution of a distributed system. In the asynchronous system model with crash failures, we introduce two new observation modalities called *negotiably* and *discernibly* (which roughly correspond to the well-known modalities *possibly* and *definitely* by Cooper, Marzullo and Neiger) and present detection algorithms for them under increasingly weak fault assumptions.

Dank

Danksagungen zeigen, daß eine Doktorarbeit das Ergebnis einer langjährigen, erfahrungsreichen persönlichen Entwicklung ist, auf die viele Menschen Einfluß genommen haben. Diese Entwicklung vollzieht sich anfangs für den Doktoranden oft unbemerkt. Erst später, im Rückblick, wird klar, wie die Dissertation einen Menschen formen kann, und daß sie im Gegensatz zur weit verbreiteten Meinung nicht das Ende der Entwicklung bildet, sondern den Beginn der Einsicht, daß diese Entwicklung nie vollendet werden kann.

An erster Stelle gilt mein Dank meinem Doktorvater Prof. Dr. Peter Kammerer, der mir in Zeiten knapper Doktorandenstellen die Promotion an der TU Darmstadt ermöglicht hat. Seine kontinuierliche Unterstützung schuf den Freiraum, in dem ich (fast) ohne äußerliche Ablenkung Ideen finden und weiterentwickeln konnte. Insbesondere bin ich für seine stetigen Ermahnungen dankbar, den "praktischen" Aspekt der Theorie nicht zu vergessen.

Prof. Dr. Friedemann Mattern danke ich für die Bereitschaft, trotz seines Wechsels an eine andere Hochschule das Korreferat dieser Arbeit zu übernehmen. Seinem guten Zureden ist es größtenteils zu verdanken, daß ich überhaupt den Schritt zur Promotion gewagt habe. Seine wissenschaftliche Arbeitsweise und seine ansteckende Begeisterung für verteilte Algorithmen haben die Gestaltung und Themenwahl dieser Arbeit stark beeinflußt. Dankbar bin ich insbesondere für die Möglichkeit, im Herbst 1999 zwei Monate als Gastwissenschaftler an der ETH Zürich zu verbringen und die vielen Vorzüge der Schweiz kennenzulernen.

Daß ich die vergangenen drei Jahre ohne finanzielle Sorgen forschen konnte, verdanke ich der Deutschen Forschungsgemeinschaft. Ich habe es als großes Privileg empfunden, Mitglied in einem ihrer Graduiertenkollegs sein zu dürfen. Die gewährte finanzielle Unterstützung und der organisatorische Rahmen des Graduiertenkollegs "Intelligente Systeme für die Informations- und Automatisierungstechnik" (ISIA) bildeten einen für mich sehr fruchtbaren wissenschaftlichen Boden.

Den Kollegen am Fachgebiet Betriebssysteme, Dr. Henning Pagnia und Dr. Oliver Theel, bin ich dankbar dafür, daß sie mich offen in ihre Arbeitsgruppe aufgenommen und mich äußerst kompetent in das weite Feld publizistischer Tätigkeit eingeführt haben. Ihre zahlreichen methodischen Anregungen haben diese Arbeit stark beeinflußt.

Zu großem Dank verpflichtet bin ich auch Gudrun Jörs, die durch ihre konstante Präsenz großen Anteil daran hatte, daß die Arbeit am Fachgebiet so viel Spaß gemacht hat. Wichtig für mich waren insbesondere am Anfang meiner Promotionszeit ihre Weigerung, in der mittäglichen Kaffeepause Fachgespräche anhören zu müssen. Geschätzt habe ich auch ihr unglaubliches Talent bei der Organi-

sation zahlreicher Fachgebietsfeiern, sowie ihre Initiative für den unvergesslichen Fachgebietsworkshop in Südfrankreich im September 1999.

Dank geht auch an die weiteren (wechselnden) Mitglieder der mittäglichen Kafferunde, vor allem Jörg Baumgart, Ralph Jansen, Udo Arnold und Wolfgang Heenes, die mich und meine Gäste vorbehaltlos in ihrer Runde akzeptierten. Obwohl es mir anfangs schwerfiel, einen Zugang zur gnadenlosen Ironie dieser mittäglichen Runde zu finden, habe ich ihre verrückte Gelassenheit, und vor allem den Wert donnerstäglicher Traditionen, sehr zu schätzen gelernt.

Einen großen Anteil am Zustandekommen dieser Dissertation hat Hagen Völzer, dem ich dafür sehr dankbar bin. Seit dem Beginn unserer langjährigen Zusammenarbeit im August 1998 habe ich seine kritischen Hinweise zu meiner Arbeit sehr geschätzt. Von ihm lernte ich Geduld, Disziplin und Rigorosität beim Formalisieren scheinbar intuitiver Begriffe, beim Aufstellen mathematischer Behauptungen und dem Durchdenken ihrer Korrektheit. Daß man in Zukunft nicht mehr "nur" nach Berlin reisen muß, um ihn zu besuchen, sondern nach Brisbane, empfinde ich als Herausforderung.

Außerordentlich inspirierend für mich war die Teilnahme an drei Seminaren im Internationalen Begegnungs- und Forschungszentrum für Informatik auf Schloß Dagstuhl im August 1998, und im März und Oktober 2000. Dankbar bin ich insbesondere für die daraus entstandenen Kontakte und anhaltenden Diskussionen mit Anish Arora und Sandeep Kulkarni, die mir gerade in der Anfangsphase meiner Arbeit viele Ideen gaben und mich später durch ihre Diskussionsbereitschaft immer wieder neu motiviert haben. Den Teilnehmern des regelmäßig stattfindenden "Diskussionskreis Fehlertoleranz", insbesondere Klaus Echtle, bin ich dankbar für ihre Offenheit für meine etwas theoretischeren Themen.

Ich habe das Vergnügen und die Ehre gehabt, in den vergangenen drei Jahren mit ausgezeichneten Kollegen zusammenzuarbeiten. Besonders gefreut hat es mich, wenn aus dieser Zusammenarbeit "schöne" Publikationen resultierten. Dankbar bin ich hierbei Sven Kloppenburg, Heiko Mantel, Henning Pagnia, Oliver Theel, Marc Theisen, Holger Vogt, Hagen Völzer, Uwe Wilhelm und Armin Wolfram. Für ihre Bereitschaft, mit mir über meine Themen zu diskutieren, bin ich zudem Matthias Bormann, Xavier Défago, Rachid Guerraoui, Ted Herman, Roger Kehr, Klaus Kursawe, Christoph Liebig, Fernando Pedone, Stefan Pleisch, Rüdiger Reischuk, Willem-Paul de Roever, André Schiper und Andreas Zeidler zu Dank verpflichtet.

Folgende Personen haben geholfen, meinen Lesehunger zu stillen, indem sie mir Kopien von Aufsätzen zukommen ließen, die in Darmstadt nicht verfügbar waren: Nuala Bennett (University of Illinois, Urbana Champaign), Felix Holderied (Universität Karlsruhe), Frau A. Matt (Mathematik-Bibliothek der Universität Marburg) und Günter Karjoth (IBM Research, Zürich).

Dirk Schlimm und Brigitte Pientka gaben mir im Juni 1999 die Gelegenheit, für ein paar Tage die Informatik-Bibliothek der Carnegie Mellon University in Pittsburgh zu benutzen. Dafür und für ihre Gastfreundschaft während dieser Zeit bin ich sehr dankbar.

Der größte Teil von Kapitel 2 entstand während meiner Zeit als Gastwissenschaftler an der ETH Zürich im Herbst 1999. Mein Dank für die freundliche Aufnahme geht hierbei an die Mitglieder der gastgebenden Arbeitsgruppe Svetlana Domnitcheva, Oliver Kasten, Marc Langheinrich und Kay Römer.

Als Probeleser stellten sich dankenswerterweise 'Verkehrsflugzeugführer A340' Uwe Lambach (Anfang Kapitel 1), Ferudun Özdemir und Christoph Liebig (Kapitel 2), Hagen Völzer (Kapitel 3) und Sven Kloppenburg (Kapitel 4) zur Verfügung.

Meinen Eltern, Kurt und Traute Gärtner, danke ich für ihr Vertrauen während der zurückliegenden Jahre, meinen "Schwiegereltern" Gerhard und Getrud Freiling für die mir entgegengebrachte Freundschaft.

Meine Verlobte Uli Freiling und ihre Katze Lina haben mir — durch bedingungslose Offenheit zum einen und durch schnurrende Gelassenheit zum anderen — geholfen, die Paradoxien dieser Welt etwas besser zu verstehen und auch in kritischen Zeiten zur Ruhe zu kommen. Ihnen verdanke ich meine körperliche und geistige Gesundheit, die in den vergangenen drei Jahren oftmals wiederhergestellt werden mußte.

Veröffentlichungen

Kapitel 3 basiert teilweise auf einer gemeinsamen Arbeit mit Hagen Völzer. Auszüge aus diesem Kapitel wurden bereits verschiedentlich veröffentlicht [73–75, 80]. Kapitel 4 basiert auf einer gemeinsamen Arbeit mit Sven Kloppenburg, die bereits in einer verkürzten Version publiziert wurde [76]. Weitere im Rahmen meiner Promotionszeit entstandene Konferenz- und Zeitschriftenveröffentlichungen zu den Themen *fair exchange* (mit Henning Pagnia, Holger Vogt und Uwe Wilhelm) [79, 145, 180], selbststabilisierende Algorithmen (mit Henning Pagnia und Oliver Theel) [77,78,173,174] und modulare Verifikation fehlertoleranter Algorithmen (mit Heiko Mantel) [132] konnten im vorliegenden Text nicht berücksichtigt werden.

Inhaltsverzeichnis

Danksagungen **7**

Veröffentlichungen **9**

1 Einführung **15**

1.1 Motivation 15
1.2 Zielrichtung dieser Arbeit 17
1.3 Aufbau der Arbeit 19
1.4 Anmerkung zu den Beweisen 20

2 Modelle für fehlertolerante Systeme **23**

2.1 Einführung 23
2.2 Grundlagen 25
2.2.1 Prozeß, Algorithmus, Berechnung 25
2.2.2 Verifikation 26
2.2.3 Reaktive Systeme und temporale Logik 27
2.2.4 Sicherheit und Lebendigkeit 29
2.3 Modell 0: Das asynchrone Systemmodell 31
2.4 Fehler, Fehlertoleranzprobleme und “FLP” 33
2.4.1 Fehlermodelle 33
2.4.2 Übereinstimmungsprobleme 36
2.4.3 Das FLP-Theorem 39
2.5 Modell 1: Einführung expliziter Zeitschranken 41
2.5.1 Synchrone Systeme 41

2.5.2 Partiell synchrone Systeme 42
2.6 Modell 2: Benutzung von Fehlerdetektoren 47
2.6.1 Perfekte Fehlerdetektoren 47
2.6.2 Unzuverlässige Fehlerdetektoren 49
2.7 Modell 3: Zeitbeschränktes asynchrones Modell 54
2.8 Modell 4: Das quasi-synchrone Modell 56
2.9 Zusammenfassung und Vergleich 58

3 Formale Grundlagen der Fehlertoleranz 61

3.1 Einführung 61
3.2 Die Fehlertoleranztheorie von Arora und Kulkarni 63
3.2.1 Detektoren und Korrektoren 63
3.2.2 Resultate 65
3.2.3 Beispiel: *triple modular redundancy* 66
3.3 Programme, Spezifikationen und Korrektheit 67
3.4 Fehlermodellierung 74
3.5 Fehlertoleranz 78
3.5.1 Definitionen von Fehlertoleranz 78
3.5.2 Fehlertolerante Versionen 80
3.6 Fehlertoleranzmechanismen für Sicherheit 82
3.6.1 Sicherheitskonservative Fehlermodelle 82
3.6.2 Tolerierbare und nicht tolerierbare Fehler 84
3.6.3 Speicherredundanz 89
3.7 Fehlertoleranzmechanismen für Lebendigkeit 92
3.7.1 Lebendigkeitskonservative Fehlermodelle 92
3.7.2 Tolerierbare und nicht tolerierbare Fehler 95
3.7.3 Zeitredundanz 97
3.8 Beispiele 99
3.8.1 Minimale Redundanz bei TMR 99
3.8.2 Zeitredundanz bei ARQ-Verfahren 101
3.9 Diskussion 104

3.10 Zusammenfassung 107

4 Fehlertolerantes Beobachten 111

4.1 Einführung 111

4.2 Fehlerfreie asynchrone Systeme 113

4.3 Fehlerbehaftete asynchrone Systeme 118

4.3.1 Bisherige Arbeiten 118

4.3.2 Prädikate und ihre Bedeutung unter der *crash*-Fehlerannahme 119

4.3.3 Beobachterabhängigkeit von *possibly* und *definitely* 121

4.3.4 Fehlerdetektoren für die Prädikatserkennung 122

4.4 Die Modalitäten *negotiably* und *discernibly* 124

4.5 Entdeckungsalgorithmen I 131

4.5.1 Zwei logische Uhren und ihre Zeitstempel 131

4.5.2 Der stabile Bereich 134

4.5.3 Algorithmus zur Entdeckung von *discernibly* 137

4.5.4 Algorithmus zur Entdeckung von *negotiably* 142

4.6 Entdeckungsalgorithmen II 144

4.6.1 Entdecken von *negotiably* bei Monitorabstürzen 144

4.6.2 Entdecken von *discernibly* bei Monitorabstürzen 150

4.7 Zusammenfassung 153

5 Zusammenfassung und Ausblick 157

Literaturverzeichnis 161

Index 179

Kapitel 1

Einführung

1.1 Motivation

Anflug auf London Heathrow. In einer Ausgabe des *Risks Digest* [109] berichtet Peter Ladkin von folgendem Vorfall: Am 19. September 1994 befindet sich ein Airbus A340 der britischen Fluggesellschaft *Virgin* auf einem Linienflug von Tokyo nach London. Bereits vor dem Abflug in Japan hatte es Probleme mit dem Computersystem gegeben, welches für die Anzeige der Flugdaten im Cockpit verantwortlich ist. Nachdem ein Steuerungscomputer zurückgesetzt worden war, schienen die Probleme jedoch behoben.

Um sicher zu gehen, daß die Fluganzeigen im Cockpit akkurat sind, veranlaßt der Flugkapitän kurz vor dem Anflug auf den Flughafen Heathrow einen zusätzlichen Abgleich der Anzeige mit einem vom Boden aus gesendeten Radiosignal. Nach wenigen Flugmeilen erlischt plötzlich die Darstellung der Flugdaten auf dem Bildschirm des Kapitäns. Es erscheint die Meldung "*please wait*" zusammen mit einer Bildschirmmaske, die normalerweise nur vor dem Start beim Laden von Flugdaten angezeigt wird. Kurze Zeit später geschieht dasselbe auf dem Bildschirm des Copiloten. Überraschenderweise ist das Flugzeug jedoch immernoch steuerbar. Dies ist bemerkenswert, da Flugzeuge vom Typ Airbus A340 nach dem *fly-by-wire*-Prinzip funktionieren. *Fly-by-wire* bedeutet, daß es keine mechanische Verbindung zwischen den Steuerungsinstrumenten der Piloten und dem Höhen- und Seitenruder der Maschine mehr gibt — alle Daten laufen durch den Computer.

Die Piloten schaffen es dennoch, den Steuerungscomputer mit dem für die Landung notwendigen Instrumentenlandesystem (ILS) zu verbinden. Als sie dies tun, gibt der Computer eine Warnung über einen zu niedrigen Treibstoffstand aus. In Absprache mit dem Tower bekommt die Maschine Priorität für eine Notlandung auf dem Flughafen mittels ILS.

Eine Landung mittels ILS erfolgt entlang eines vom Boden aus gesendeten Funk-

strahls, der einen Anflugwinkel von etwa 3 bis 4 Grad vorgibt. Es ist bekannt, daß es technisch bedingt neben dem Hauptstrahl mehrere “falsche Seitenstrahlen” gibt, die man jedoch bei einem normalen Anflug nicht trifft. Die Piloten haben in dieser Situation jedoch einen falschen *radar vector* von der Flugleitzentrale erhalten, so daß der Autopilot beim Anflug einen solchen falschen Funkstrahl erfaßt. Das Flugzeug ändert seinen Anflugwinkel auf etwa 9 Grad. Der Flugkapitän schaltet die Verbindung des Autopiloten mit dem ILS ab und beantragt einen *surveillance radar approach* (SRA), bei dem in einem ununterbrochenen Strom die notwendigen Instrumenteninformation durch den Fluglotsen im Tower über Funk an den Piloten durchgegeben werden.

Während des SRA muß eine Linkskurve von etwa 90 Grad gefolgen werden und die Piloten drehen die Richtungsvorgabe am Autopiloten auf den entsprechenden Wert. Statt nach links zu fliegen, beginnt das Flugzeug jedoch plötzlich nach rechts zu steuern. Die Piloten schalten daraufhin alle automatischen Landehilfen ab und steuern das Flugzeug im “Handbetrieb” weiter. Die Landung erfolgt ohne weitere Zwischenfälle.

Analyse des Zwischenfalls. Die Analyse der Geschehnisse durch die britischen Behörden stellte fest, daß eine Kombination mehrerer Faktoren für den Zwischenfall verantwortlich waren. Die falsche Reaktion des Autopiloten während der Linkskurve hatte ihre Ursache in Software-Fehlern, die bei anschließenden *upgrades* korrigiert worden sind. Die Warnung über mangelnden Treibstoff stellte sich als falsch heraus und wurde auf die Tatsache zurückgeführt, daß während des Fluges Treibstoff zwischen den verschiedenen Tanks im Flugzeug hin- und hergepumpt wird, um den Schwerpunkt des Flugzeuges konstant zu halten. Es wurde vorgeschlagen, fünf zusätzliche Treibstoffsensoren pro Tank zu installieren und die Software entsprechend anzupassen.

Der Grund für den Ausfall beider Fluganzeigen konnte jedoch nicht genau identifiziert werden. Die Analyse der Protokolldateien ergab lediglich, daß beide Anzeigesysteme zu etwa der gleichen Zeit einen Fehler der Kategorie “*class 1 hard*” meldeten. Derartige Fehler sind bereits bei Flugzeugen der Klasse A320 aufgetreten und es wird vermutet, daß dafür eine Kombination aus einem seltenen Hardwarefehler und einer unerwünschten Beeinflussung des einen Computers durch den anderen ursächlich ist.

Die Abhängigkeit von zuverlässigen Computersystemen. Der eingangs geschilderte Zwischenfall mit dem Airbus A340 ist ein paradigmatisches Beispiel dafür, wie stark Computersysteme bereits heute die Gesellschaft durchdrungen haben und wie stark unser aller Wohlergehen von ihnen abhängt. Das Buch “*Computer Related Risks*” von Peter G. Neumann [141] enthält die Beschreibung einer Reihe weiterer Zwischenfälle aus vielen verschiedenen Bereichen, vor allem

den sogenannten *kritischen Infrastrukturen* [97, 164]. Dazu zählen unter anderem die Verkehrssysteme (Flugzeuge, Züge, Schiffe), das Gesundheitswesen, die Finanzwelt und die Energieversorgung. Viele Anzeichen deuten darauf hin, daß Computer auch unseren gewöhnlichen Lebensalltag so stark durchdringen werden, daß sie allgegenwärtig (*ubiquitous*), also kaum noch explizit wahrnehmbar sind. Wenn man die Probleme bedenkt, die bereits heute dadurch entstehen, daß in einem Haushalt der Strom oder das Telefon ausfällt, wird deutlich, daß die Abhängigkeit von funktionierenden Computersystemen in Zukunft eher zu- als abnehmen wird. Die zwangsläufig auftretenden Probleme und die dadurch verursachten Zwischenfälle resultieren nicht nur in Chaos und Konfusion, sondern können Schäden in Millionenhöhe bis hin zum Verlust von Menschenleben verursachen. Dennoch rechtfertigt die Flexibilität, die man durch die Einführung von Computern auch in sensiblen Bereichen der Gesellschaft erreicht, die damit einhergehenden Risiken in vielen Fällen. Manche Anwendungen, wie z.B. die Steuerung von aerodynamisch instabilen Düsenflugzeugen, sind ohne Computerunterstützung gar nicht mehr möglich.

Computersysteme, die innerhalb von kritischen Infrastrukturen eingesetzt werden, bergen ein großes Risiko: Ein Ausfall führt in der Regel zu hohen Schäden. Die Besorgnis über die damit einhergehenden Gefahren hat zu einer breiten Diskussion in der Fachöffentlichkeit geführt, die durch eine zunehmende Anzahl von Sonderheften zu diesem Thema [26, 100] dokumentiert ist. Sogenannte *kritische Computersysteme* müssen *verläßlich* (*dependable*) sein. Verläßlichkeit bedeutet, daß man sich berechtigterweise auf das korrekte Funktionieren des Systems verlassen kann [116]. Verläßlichkeit definiert sich aus einer Kombination von Eigenschaften wie *Zuverlässigkeit* (*reliability*), *Sicherheit* (*safety*), *Verfügbarkeit* (*availability*) und *Integrität* (*integrity*) [98, 116, 141]. Die Gesellschaft gibt für kritische Computersysteme Verläßlichkeitsanforderungen vor (oft in Form von Gesetzen oder Standards [29]). Beispielsweise muß die Wahrscheinlichkeit eines kritischen Fehlers im Steuerungssystem eines Verkehrsflugzeuges pro Flugstunde weniger als 10^{-9} liegen [155]. Während des Entwurfs, bei der Konstruktion und während des Betriebs müssen diese Anforderungen validiert werden, d.h. Fachgutachter müssen zu der Überzeugung gelangen, daß das System die Anforderungen erfüllt.

1.2 Zielrichtung dieser Arbeit

Ingenieursmethoden zu entwickeln, mit denen verläßliche Systeme konstruiert und validiert werden können, ist eine Aufgabe der Wissenschaft. Je höher die Verläßlichkeitsforderungen und je größer die Systeme, desto aufwendiger und komplexer werden naturgemäß derartige Konstruktions- und Validierungsmethoden. Dies gilt insbesondere für *verteilte Systeme*, wie zum Beispiel Rechner-

netze. Verteilte Systeme sind dadurch gekennzeichnet, daß eine Vielzahl von unabhängigen aktiven Komponenten oft geographisch verteilt zusammenarbeiten und Informationen über ein zwischengeschaltetes Kommunikationsmedium austauschen. Die Beherrschung praktischer verteilter Computersysteme ist eine der großen Herausforderungen des 21. Jahrhunderts.

Große Systeme, wie der weltweit vernetzte Aktienhandel, die Steuerung des Flugverkehrs oder das Internet stoßen heute an Grenzen, an denen einzelne Personen oder Personengruppen kaum noch verstehen, wie oder warum das System funktioniert. Im Gegensatz zu Einzelplatzcomputern oder eng gekoppelten Parallelrechnern sorgen bei verteilten Systemen die nichtdeterminstische Parallelität im Programmablauf sowie die Verteiltheit der Daten zu einer Komplexität, der heutige Ingenieursmethoden kaum noch gewachsen sind. Verteilte Systeme sind jedoch in einem gewissen Sinne auch allgemeiner als eng gekoppelte Systeme, so daß allgemeine Lösungen für verteilte Systeme sich oft in vereinfachter Form auf nicht-verteilte Systeme übertragen lassen.

Diese Arbeit versteht sich als ein Beitrag zur Konstruktionsmethodik verläßlicher Systeme, d.h. als ein Schritt auf dem Weg zur Beherrschung komplexer verteilter Systeme. Die Erfahrung lehrt, daß eine solche Methodik immer die ganzheitliche Betrachtung des Entwicklungszusammenhanges erfordert. Zwischenfälle (wie der eingangs geschilderte) haben nahezu immer *mehrere* Ursachen, die heutzutage zum großen Teil an mangelhafter Software liegen [141]. In verteilten Systemen sind jedoch oft auch Hardware-Fehler der Auslöser für unerwünschtes Verhalten. In dieser Arbeit beschränke ich mich darum auf das Gebiet der *Fehlertoleranz* in verteilten Systemen. Fehlertoleranz meint, daß ein Computersystem trotz einer begrenzten Menge interner Fehler weiterhin das erwartetes Verhalten aufzeigt. Oft wird ironisierend behauptet, Computer lösten Probleme, die es ohne Computer nicht gäbe. Die Fehlertoleranz ist ein Gebiet, auf das diese Aussage uneingeschränkt zutrifft.

In dieser Arbeit sind drei Arbeitspakete gesammelt, die ich während meiner Promotionszeit bearbeitet habe. Sie spannen den Bogen von theoretischen Modelluntersuchungen (Kapitel 2), über Terminologiefragen (Kapitel 3) bis hin zu konkreten Algorithmen und Fehlertoleranzmechanismen (Kapitel 4). Den Kapiteln gemeinsam ist ein Erkenntnisinteresse, das nicht unmittelbar auf eine praktische Verwertbarkeit abzielt, aber dennoch den Weg ebnen sollte zu einem besseren Verständnis praktischer Systeme sowie zu verbesserten Methoden zu ihrer Beherrschung.

1.3 Aufbau der Arbeit

Fehlertolerante verteilte Systeme bilden den Hauptuntersuchungsgegenstand dieser Arbeit.

Der Fokus von Kapitel 2 liegt auf einer vergleichenden Modellanalyse für fehlertolerante verteilte Systeme. Reale Netzwerke sind sehr komplex und man benötigt oft ein präzises Modell dieser Netzwerke, um die darin ablaufenden Systeme validieren zu können. Modelle beschreiben eine Menge von Annahmen über das reale System. Mittels dieser Annahmen ist es möglich, Eigenschaften von Systemen formal nachzuweisen. In der Fehlertoleranz erstrecken sich die Modellannahmen auch auf Fehler und mögliches Fehlverhalten im System. In Kapitel 2 werden zunächst die Grundbegriffe (System, Fehler, Eigenschaft, Fehlertoleranz) auf einem eher informalen Weg eingeführt. Für verteilte Systeme, die trotz Fehlern noch korrekt funktionieren sollen, existieren mindestens vier prominente Modellvorschläge, die in der Literatur jedoch nur sehr technisch beschrieben worden sind. Kapitel 2 versucht, in einem eher informalen, didaktischen Ansatz diese Modelle vorzustellen und ihre jeweiligen Vor- und Nachteile bei der Beschreibung realer Systeme herauszuarbeiten. Abgesehen von einer etwas zugänglicheren Darstellung des Themenbereiches liegt der Hauptbeitrag von Kapitel 2 darin, daß es im Anwendungsfall einfacher wird, sich für das "richtige" Modell zu entscheiden.

Kapitel 3 baut auf der in Kapitel 2 eher informal eingeführten Begrifflichkeit auf und entwickelt ein vollständig formales Systemmodell fehlertoleranter Systeme. Das Modell basiert im wesentlichen auf bereits existierenden Vorarbeiten von Arora und Kulkarni, die innerhalb dieses Modells eine allgemeine Theorie fehlertoleranter Systeme vorgestellt haben [14, 15]. Wie an anderer Stelle bereits gezeigt [74], eignet sich diese Theorie gut sowohl zur Gliederung des Gebietes der Fehlertoleranz als auch zur einfachen Klassifikation von Fehlertoleranzmethoden. Im Hauptteil von Kapitel 3 wird diese Theorie um eine formale Begrifflichkeit von *Redundanz* erweitert und es wird gezeigt, daß bestimmte Formen von Redundanz notwendig sind, um bestimmte Fehlertoleranzeigenschaften zu erreichen. Die Definitionen von Redundanz und die Notwendigkeitsresultate sind grundlegend und neu. Wie in Kapitel 3 gezeigt wird, erlauben sie erstmals Fehlertoleranzverfahren bezüglich ihrer Redundanz zu vergleichen und entsprechende Optimalitätsbetrachtungen anzustellen. Außerdem geben die Resultate Antworten auf die Frage, warum und wie die Fehlertoleranztheorie von Arora und Kulkarni "funktioniert".

Kapitel 4 beschäftigt sich wieder konkret mit Lösungen von Problemen, die durch die Verteiltheit großer Systeme entstehen. Ein Problem von praktischer Relevanz ist die Entdeckung von globalen Prädikaten in verteilten Systemen. Instanzen dieses Problems sind das Erkennen der verteilten Terminierung, *deadlock*-Erkennung oder die Überprüfung von unerlaubten Zuständen beim verteilten *debugging*. Bisherige Algorithmen für diese Probleme funktionierten größtenteils

nur in fehlerfreien Systemen. Aufbauend auf dem in Kapitel 2 eingeführten *Fehlerdetektormodell* werden in Kapitel 4 zwei Beobachtungsbegriffe für asynchrone verteilte Systeme, in denen einzelne Rechnerknoten abstürzen können, definiert. Die Definitionen sind neu und in ihrer Semantik "natürliche" Erweiterungen von existierenden Begriffen für fehlerfreie Systeme. In Kapitel 4 werden außerdem Erkennungsalgorithmen für die neu eingeführten Beobachtungsbegriffe entwickelt.

Kapitel 5 faßt die grundlegenden Resultate dieser Arbeit nochmals kurz zusammen und gibt einen Ausblick auf Themen und Fragestellungen, die sich als weitere Arbeitsgebiete aus dieser Dissertation ergeben haben.

Wie kann man diese Arbeit lesen? Mögliche Lesepfade dieser Arbeit sind in Abbildung 1.1 dargestellt. Kapitel 2 liefert das terminologische Grundgerüst für die folgenden Kapitel und sollte deswegen zumindest bis Abschnitt 2.4 gelesen werden. Kapitel 3 und 4 sind weitestgehend unabhängig lesbar. Für Kapitel 4 sollte jedoch Kapitel 2 mindestens bis Abschnitt 2.6 gelesen worden sein. Für eilige Leser sind zu Beginn und Ende der Kapitel die Grundgedanken jeweils kurz zusammengefaßt.

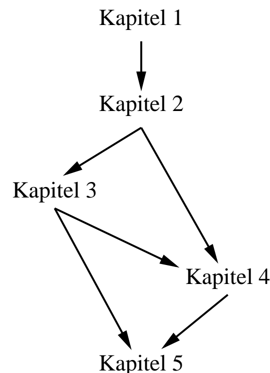


Abbildung 1.1: Mögliche Lesepfade dieser Arbeit.

1.4 Anmerkung zu den Beweisen

In den Kapiteln 3 und 4 werden über die Korrektheit von bestimmten Sätzen Beweise geführt. Die Darstellung der Beweise erfolgt in einer strukturierten Notation ähnlich der von interaktiven Theorembeweisern. Diese Form wurde von Lamport [114] vorgeschlagen, der behauptet, daß dieser Stil es sehr viel schwerer mache, Sätze zu beweisen, die falsch sind. Ein Beweis ist eine Folge von numerierten Schritten auf verschiedenen Ebenen. Jeder Schritt enthält entweder eine kurze Begründung, warum er gültig ist, oder er wird wieder selbst zu einer Folge

von Schritten verfeinert. Die Numerierung folgt der Struktur. Beispielsweise ist Schritt $\langle 1 \rangle 2.$ der zweite Schritt auf Stufe 1. Strukturierte Beweise sehen auf den ersten Blick relativ furchteinflößend aus. Durch die gewählte Darstellungsform ist es jedoch möglich, Beweise selektiv zu lesen, indem man Schritte auf niedrigeren Ebenen nur liest, wenn es nötig erscheint. In der Regel wird zu Beginn eines Beweises jeweils eine Beweisskizze gegeben.

Kapitel 2

Modelle für fehlertolerante verteilte Systeme

2.1 Einführung

Grundprobleme verteilter Systeme. Verteilte Systeme sind dadurch gekennzeichnet, daß viele unabhängige Einheiten (Prozesse, Prozessoren, Computer) existieren, deren einzige Kommunikationsmöglichkeit darin besteht, Informationen mittels Nachrichten über ein zwischen ihnen bestehendes Kommunikationsnetzwerk auszutauschen. Im Gegensatz zu eng gekoppelten Parallelrechnern oder herkömmlichen Ein- oder Mehrprozessorsystemen entstehen für Algorithmen in verteilten Systemen dadurch drei wesentliche Probleme:

- Das Fehlen einer gemeinsamen Zeit.

  Eine gemeinsame "globale Zeit," wie man sie aus dem täglichen Leben kennt, ist in verteilten Systemen nicht in beliebiger Genauigkeit realisierbar. Bei einer hinreichenden Größe des Netzwerkes ist es unmöglich, die Prozesse durch eine einzelne Uhr mit einem gemeinsamen Takt zu versorgen. Man muß davon ausgehen, daß die Uhren der einzelnen Prozesse nicht beliebig genau synchronisiert werden können und daß nur ungenaue Aussagen über die Ausführungsgeschwindigkeiten verschiedener Prozesse relativ zueinander gemacht werden können.

- Das Fehlen einer globalen Sicht.

  Es ist bei hinreichender Größe des verteilten Systems unmöglich, aus einer einzigen Beobachtungsposition den Zustand des Gesamtsystems zu einem beliebigen Zeitpunkt sofort und als Ganzes zu erfassen. Dies macht Aussagen der Form "Das System befindet sich jetzt im Zustand x" unmöglich

und hat weitreichende Auswirkungen auf Algorithmen, die den Zustand des Gesamtsystems betrachten oder darüber Aussagen machen sollen.

- Der inhärente Nichtdeterminismus.

  Obwohl die einzelnen Programme auf den Prozessoren alle vollkommen determiniert sein können, ist das Verhalten des gesamten verteilten Systems im allgemeinen trotzdem nichtdeterministisch. Dies liegt einerseits an den unbestimmten Laufzeiten der Nachrichten im zugrundeliegenden Kommunikationsnetzwerk, andererseits an der inhärenten Parallelität des Systems und dem fehlenden gemeinsamen Bezugspunkt "Zeit."

Die Beschäftigung mit verteilten Systemen und deren Algorithmen stellt die Forscher also vor eine vollkommen neue Klasse von Problemen, die bei herkömmlichen Rechnersystemen kaum eine Bedeutung haben. Die verteilten Systeme sind geradezu eine "Verallgemeinerung" der herkömmlichen und bereits gut erforschten sequentiellen und parallelen Systeme.

Die Notwendigkeit eines präzisen Modells. Wegen der oben beschriebenen Probleme sind verteilte Algorithmen nicht dadurch begreifbar, daß man sich nacheinander und jeweils einzeln die Programme auf den jeweiligen Rechnerknoten anschaut. Immer wieder sind deshalb auch inkorrekte Algorithmen veröffentlicht worden (beispielsweise zur Erkennung der verteilten Terminierung [18]; vergleiche hierzu auch die Gegenbeispiele [171] und Erwiderungen [17]). Ein sehr einfacher Algorithmus, der vielleicht nur aus einem halben Dutzend Befehlen besteht, wirkt plötzlich sehr verwirrend, wenn er auf mehreren Prozessoren gleichzeitig ausgeführt wird und zu keinem Zeitpunkt klar ist, welcher Prozessor den nächsten Schritt macht [126, S. 3]. Auf der anderen Seite ist es auch sehr schwierig, Voraussagen über die Wirkungsweise eines Algorithmus zu machen, indem man sich anschaut, wie er in einem konkreten Fall abläuft. Bei einem erneuten Durchlauf desselben Algorithmus wird er möglicherweise vollkommen anders ablaufen, auch wenn sich die Eingaben überhaupt nicht verändert haben. All dies macht es notwendig, verteilte Algorithmen *sehr genau zu beschreiben* und ihren Ablauf in einem *präzisen Modell* zu verankern, um damit Aussagen über Eigenschaften des Algorithmus selbst und nicht eines konkreten Ablaufes zu ermöglichen.

Modelle als Sichten der Wirklichkeit. Grundlage der Modellbildung ist die Abstraktion. Modelle sind eine abstrakte Repräsentation konkreter Systeme aus unserer täglichen Erfahrungswelt (der "Realität" oder "Wirklichkeit") und unterliegen — wie alle Formen der Abstraktion — auch einer Interpretation, welche Eigenschaften des Systems wesentlich und welche unwesentlich sind. (Die unwesentlichen Eigenschaften werden "abgezogen", von lat. *abstrahere* = abziehen.) Lamport und Lynch charakterisieren solche Modelle auch mit unterschiedlichen

Sichten (*views*) der Wirklichkeit [115, S. 1159]. Abstraktion ist notwendig, weil das, was wir mit “Realität” bezeichnen, selbst in Ausschnitten zu komplex ist, um es wirklich präzise, vollständig und übersichtlich zu beschreiben.

Ziel des Kapitels. Die Notwendigkeit der Abstraktion impliziert, daß es kein “bestes Modell” geben kann; jedes wird seine eigenen Stärken und Schwächen haben und sich unterschiedlich gut für bestimmte Anwendungsgebiete eignen. Dies trifft auch auf den Bereich der Fehlertoleranz zu. In diesem Kapitel werden die wichtigsten Modelle für fehlertolerante, verteilte Systeme vorgestellt, miteinander verglichen und ihre Vor- und Nachteile im Bezug auf die Beschreibung bestimmter Klassen von fehlertoleranten Systemen untersucht. Diese Übersicht ermöglicht es, sich für den konkreten Anwendungsfall jeweils für das am besten geeignete Modell zu entscheiden. Die Darstellung erfolgt in einer eher informalen Weise und legt darum eine Basis für das anschliessende Kapitel 3, in dem die Begriffe im Rahmen einer allgemeinen Theorie formaler präzisiert werden.

Aufbau des Kapitels. Nach einer kurzen Einführung der wichtigsten Terminologie in Abschnitt 2.2 wird das sogenannte asynchrone Systemmodell in Abschnitt 2.3 vorgestellt. Im darauffolgenden Abschnitt 2.4 wird dieses Modell im Hinblick auf seine Verwendbarkeit im Rahmen von Fehlertoleranzuntersuchungen evaluiert. Hier stellt sich heraus, daß dieses Systemmodell neben einigen Vorteilen auch einen entscheidenden Nachteil hat. Die in den Abschnitten 2.5 bis 2.8 vorgestellten, erweiterten Systemmodelle versuchen, diesen Nachteil durch unterschiedliche zusätzliche Annahmen zu beseitigen. Die Art und Weise, in der sie dies tun, führt dazu, daß sie für ganz unterschiedliche Einsatzgebiete geeignet sind. Dies wird im zusammenfassenden Abschnitt 2.9 erörtert.

2.2 Grundlagen

2.2.1 Prozeß, Algorithmus, Berechnung

Obwohl man mit einem verteilten System oft ein Rechnernetz meint, in dem einzelne Computer räumlich verteilt zusammenarbeiten, hat es sich eingebürgert, die aktiven Elemente eines verteilten Systems mit dem Begriff *Prozeß* zu bezeichnen. Dies trägt der Tatsache Rechnung, daß auf den Einzelrechnern zumeist viele einzelne Aktivitätsträger in Form von Prozessen (*processes*) oder Leichtgewichtsprozessen (*threads*) auf der Ebene des Betriebssystems existieren, die alle jeweils einzeln an unterschiedlichen Aufgaben arbeiten und sich an der Kommunikation innerhalb des Gesamtsystems beteiligen. Zwischen den Prozessen existieren *Kommunikationskanäle*, über die Informationen in Form von Nach-

richten ausgetauscht werden. Dazu existieren spezielle Sende- und Empfangsoperationen. Wenn es nicht auf die Kommunikationstopologie ankommt, werden die Kommunikationskanäle oft zusammengefaßt zu einem *Kommunikationssubsystem*, welches das Senden von Nachrichten von jedem Prozeß zu jedem anderen Prozeß ermöglicht.

Informal kann man sich einen Prozeß wie ein normales (sequentielles) Programm vorstellen (mit Variablen, Programmzähler, *stack* usw.). Der *lokale Zustand* (*local state*) eines Prozesses ist eine Belegung der Variablen mit konkreten Werten. Hierbei werden Programmzähler und *stack* wie normale Variablen betrachtet, d.h. ein Zustand ist eine Art Abbild des veränderlichen Speichers eines Prozesses. Der Zustand des Kommunikationssubsystems wird üblicherweise definiert als die Menge der Nachrichten, die abgeschickt aber noch nicht empfangen worden sind.

Da die Betrachtungen dieser Arbeit unabhängig von einer konkreten Rechnerarchitektur sind, wird in dieser Arbeit statt "Programm" der Begriff "Algorithmus" verwendet. Ein *lokaler Algorithmus* (*local algorithm*) ist eine Abarbeitungsvorschrift, die beschreibt, wie sich *ein einzelner Prozeß* zu verhalten hat. Im wesentlichen besteht ein lokaler Algorithmus aus einer Beschreibung der erlaubten Zustandsübergänge des Prozesses. Wird der Algorithmus gestartet und wird er von einem Prozeß abgearbeitet, spricht man von einer *lokalen Berechnung* (*local execution*). Eine Berechnung ist demnach eine Folge von Zuständen, die der Prozeß durchläuft.

Ein *globaler Zustand* (*global state*) eines verteilten Systems ist eine Menge von lokalen Zuständen der beteiligten Prozesse plus dem Zustand des Kommunikationssubsystems. Ein *verteilter Algorithmus* ist eine Menge von lokalen Algorithmen, die ihre Zustände nur mittels Nachrichten synchronisieren können. Ein *Ablauf* (*trace*, *execution*) des verteilten Algorithmus ist eine Folge von globalen Zuständen des verteilten Systems. Aufgrund des inhärenten Nichtdeterminismus eines solchen Systems kann es vorkommen, daß derselbe verteilte Algorithmus ausgehend aus dem gleichen globalen Zustand unterschiedliche Abläufe verfolgt. Die Menge aller derartig möglichen Abläufe ist für den Algorithmus charakteristisch [59]

2.2.2 Verifikation

Validierung, Verifikation, Spezifikation, Korrektheit. In der Praxis ist es notwendig und oft sogar gesetzlich vorgeschrieben, daß bestimmte Systeme bezüglich ihrer Verläßlichkeit bewertet werden. Dies wird als *Validierung* bezeichnet. Ein Teilgebiet der Validierung ist die *Verifikation*. Mit Verifikation bezeichnet man die Tätigkeit, bei der man (salopp gesagt) zeigt, daß ein System genau das tut, was es tun soll (und nichts anderes). Die erwünschten Eigenschaften des Systems werden in einer sogenannten *Spezifikation* (*specification*)

festgelegt. Zum Beispiel beschreibt die Spezifikation einer Transaktion in einem replizierten Buchungssystem, daß die Buchung entweder ganz (d.h. überall) oder gar nicht (d.h. nirgends und ohne Seiteneffekte) ausgeführt wird. Wenn ein System nachweislich nur das tut, was in der Spezifikation beschrieben worden ist, dann wird es als *korrekt* bezeichnet.

Eine Spezifikation beschreibt gewünschte Eigenschaften, also impliziert sie auch das folgende Problem: Finde einen Algorithmus, der diese Spezifikation innerhalb eines bestimmten Systemmodells erfüllt. Manchmal wird dies (gerade im Bereich der Fehlertoleranz) nicht möglich sein. Wenn gewisse Fehler beispielsweise dazu führen können, daß ein Prozeß spontan in einen Zustand "Transaktion erfolgreich beendet" springen kann, dann ist es ohne weiteres nicht möglich, einen Algorithmus zu realisieren, der korrekt ist bezüglich der Spezifikation einer Transaktion.

Qualitative vs. quantitative Eigenschaften. Der Fokus in dieser Arbeit liegt auf der Untersuchung *qualitativer* Eigenschaften eines verteilten Systems und nicht auf *quantitativen Eigenschaften*. Quantitative Eigenschaften beschreiben beispielsweise, nach wieviel Sekunden ein Programm terminiert, wie oft pro Sekunde eine bestimmte Instruktion ausgeführt wird oder bis wann spätestens eine Operation ausgeführt worden sein muß. Qualitative Eigenschaften hingegen sagen lediglich aus, *daß* ein Algorithmus schließlich terminiert oder daß eine bestimmte Anweisung im Laufe der Berechnung *beliebig oft* ausgeführt wird. Der Fokus auf qualitative Eigenschaften wurde durch die Definition eines Ablaufes als Folge von Zuständen bereits in die Grundfesten des hier entstehenden Systemverständnisses eingebaut. In diesem Verständnis ist eine *Eigenschaft* formal definiert als eine Menge von Abläufen eines Systems. Wenn ein System nur Abläufe aufzeigt, die "Element von" einer Eigenschaft sind, dann besitzt es diese Eigenschaft.

In der Praxis möchte man natürlich sowohl qualitative als auch quantitative Eigenschaften validieren. Die Betrachtung qualitativer Eigenschaften ist demnach nur ein erster Schritt. In der Praxis macht es jedoch keinen Sinn, wenn man gleich die quantitativen Eigenschaften eines Systems untersucht, ohne sich über die qualitativen Eigenschaften sicher zu sein.

2.2.3 Reaktive Systeme und temporale Logik

Ein etablierter Teil moderner Informatik-Curricula ist das Gebiet der *Programmverifikation*. Begriffe wie *Vorbedingung*, *Nachbedingung*, *Zusicherung* und *Invariante* gehören mittlerweile zum normalen Sprachinventar eines Informatikers. Die oft wenig hinterfragte Grundannahme bei der standardmäßigen Behandlung dieses Gebietes ist, daß ein Programm im wesentlichen eine *Transformation* be-

schreibt. Ein Programm zur Sortierung einer Namensliste beispielsweise erhält als Eingabe eine ungeordnete Liste von Namen und verändert sie so, daß die Namen nach Beendigung des Algorithmus in alphabetischer Reihenfolge sortiert sind. Die Korrektheitsbedingungen lassen sich demnach zweiteilen in

1. partielle Korrektheit (formuliert als *Vorbedingung* und *Nachbedingung*, *precondition/postcondition*), und

2. Terminierung (d.h. der Algorithmus hält nach endlicher Zeit).

Zu den Methoden, wie man nachweisen kann, daß ein Algorithmus tatsächlich diesen Bedingungen genügt, existieren viele auch heute noch lesenswerte Klassiker der Informatik-Literatur [47, 82].

Mit dem Aufkommen verteilter Systeme wurde aber schnell klar, daß es auch Algorithmen gibt, deren Korrektheitsbedingungen sich nicht so einfach mittels eines Ein-/Ausgabeverhaltens beschreiben lassen. Dies wird vor allem deutlich, wenn man *Betriebssysteme* oder *Netzwerkprotokolle* betrachtet. Derartige *reaktive Systeme* sind dadurch gekennzeichnet, daß sie eine *nichtterminierende* Berechnung ausführen, die eine kontinuierliche Interaktion mit der Umgebung (Benutzern oder anderen Prozessen) aufrechterhalten. Das *network time protocol* (NTP), das zur Uhrensynchronisation im Internet eingesetzt wird, besteht beispielsweise aus einem kontinuierlichen Austausch von Nachrichten und einer sukzessiven aber nie perfekten Annäherung der einzelnen Uhrzeiten auf vernetzten Computern [139].

Zur Beschreibung der Eigenschaften reaktiver Systeme hat sich die *temporale Logik* [129,130,149] als sehr nützlich erwiesen. Temporale Logik ist eine Spezialform modaler Logik [59], mit der man beschreiben kann, wie sich der Wahrheitswert bestimmter Zusicherungen über die Zeit verändert. Die grundlegenden Operatoren der temporalen Logik sind “$\Diamond$” (zu lesen als “*eventually*”, deutsch “nach endlicher Zeit”) und “$\Box$” (“*always*”, “immer”). Die Formel $\Box\varphi$ ist genau dann wahr, wenn der Wert des Prädikates φ ab jetzt und für immer den Wert *true* hat. Umgekehrt ist die Formel $\Diamond\varphi$ genau dann wahr, wenn es mindestens einen Zeitpunkt in der Zukunft gibt, an dem das Prädikat φ den Wert *true* annimmt (danach kann es wieder *false* werden). In den Formeln kann φ statt eines Prädikates auch eine ganz normaler Term propositionaler Logik oder selbst wieder temporale Operatoren beinhalten. Beispielsweise bedeutet die Formel

$$\Box(\alpha \Rightarrow \Diamond\beta)$$

daß wann immer α gilt, nach endlicher Zeit dann auch β gilt. Wenn α beispielsweise den Empfang einer Suchanfrage darstellt und β das Senden der entsprechenden Antwort, bedeutet diese Formel etwa “jede Anfrage wird bearbeitet”. Die herkömmlichen Korrektheitsbedingungen sequentieller Programme können auch mit Hilfe temporaler Logik beschrieben werden:

- Wenn *PRE* und *POST* die Vor- und Nachbedingungen eines Programmes beschreiben und *INIT* und *TERM* Prädikate sind, die jeweils im Anfangszustand und im Terminierungszustand des Programmes gelten, dann drückt die Formel

  $$INIT \wedge PRE \Rightarrow \Box(TERM \Rightarrow POST)$$

  die partielle Korrektheit aus.

- Auf der anderen Seite drückt dann

  $$\Diamond TERM$$

  die Terminierung aus.

2.2.4 Sicherheit und Lebendigkeit

Es gibt im wesentlichen zwei große Klassen von Eigenschaften reaktiver Systeme: Sicherheitseigenschaften und Lebendigkeitseigenschaften [111].

Eine *Sicherheitseigenschaft* (*safety property*) beschreibt salopp, daß irgendetwas Unerwünschtes nie eintreten wird ("Something bad will never happen." [111]). Beispiele für Eigenschaften dieser Klasse sind der wechselseitige Ausschluß oder die herkömmliche partielle Korrektheit von sequentiellen Programmen. Genauer besagt eine Sicherheitseigenschaft, daß ein gewisses Prädikat in jedem erreichbaren globalen Zustand (quasi "immer" [176]) gilt [12, S. 182]. Nachgewiesen werden solche Eigenschaften in der Regel durch das Schließen mit Zusicherungen (*assertional reasoning*). Eine Zusicherung ist in diesem Zusammenhang ein Prädikat P auf Systemzuständen und man muß zeigen, daß dieses Prädikat durch die Aktionen des Algorithmus nicht verletzt wird. Für P benutzt man auch häufig den Ausdruck "Invariante."

Auf der anderen Seite beschreibt eine *Lebendigkeitseigenschaft* (*liveness property*), daß irgendwann etwas Erwünschtes eintreten wird ("Something good will eventually happen." [111]). Das Paradebeispiel für eine solche Eigenschaft ist die Terminierung eines Algorithmus. Lebendigkeitseigenschaften sind etwas schwieriger zu verstehen als Sicherheitseigenschaften: Beispielsweise ist die Verklemmungsfreiheit ("Es ist nie der Fall, daß sich das System in einer Verklemmungssituation befindet.") eine Sicherheitseigenschaft und die Aushungerungsfreiheit ("Wenn ein Prozeß eine Ressource beantragt, erhält er sie auch nach endlicher Zeit.") eine Lebendigkeitseigenschaft. Lebendigkeitseigenschaften besagen, daß keine Ausführung eines Algorithmus "für immer verloren" ist; es muß immer möglich sein, daß das besagte erwünschte Ereignis in Zukunft noch passieren wird. Falls dies nicht mehr möglich ist, könnte man dies an einem gewissen Punkt festmachen und hätte so etwas Unerwünschtes beschrieben. Deshalb können *liveness*-

Eigenschaften nie etwas Unerwünschtes beschreiben und sind in gewisser Weise also komplementär zu den *safety*-Eigenschaften. (Mehr dazu in Kapitel 3.)

Der Beweis von *liveness*-Eigenschaften erfolgt in der Regel analog zur Terminierung bei der sequentiellen Programmierung durch ein Wohlfundiertheitsargument. Dafür konstruiert man eine Funktion (die Terminierungsfunktion oder *norm function* [172, S. 53]) aus der Menge der Konfigurationen in eine wohlfundierte Menge (beispielsweise die natürlichen Zahlen) und zeigt, daß in jeder Ausführung die Funktion streng monoton fällt oder die geforderte Bedingung (zum Beispiel Terminierung) bereits gilt.

Zusammenhang zwischen *safety* und *liveness*. In ihrem Artikel "Defining Liveness" [12] zeigen Alpern und Schneider, daß jede nicht-triviale Eigenschaft eines Algorithmus als die Schnittmenge einer *safety*- und einer *liveness*-Eigenschaft ausgedrückt werden kann. Das bedeutet, daß zum Nachweis der Korrektheit verteilter Algorithmen in der Regel zwei Teilbeweise notwendig sein werden. Auf der anderen Seite hilft diese Feststellung auch bei der Spezifikation reaktiver Systeme. Beispielsweise wird manchmal das Problem des wechselseitigen Ausschlusses (*mutual exclusion problem*) allein wie folgt spezifiziert:

- Es ist nie der Fall, daß sich zwei Prozesse zur gleichen Zeit innerhalb ihres kritischen Abschnittes befinden.

Diese Eigenschaft ist offensichtlich eine Sicherheitseigenschaft. Sicherheitseigenschaften allein sind aber recht einfach zu erfüllen, nämlich durch ein Programm, welches "nichts" tut. Um derartige triviale Lösungen auszuschließen, braucht man noch eine zusätzliche Eigenschaft, die wie folgt definiert werden könnte:

- Wenn ein Prozeß den kritischen Abschnitt betreten will, so gelingt ihm dies auch nach endlicher Zeit.

Erst durch diese zusätzliche *liveness*-Bedingung wird das Problem des wechselseitigen Ausschlusses interessant. Man beachte, daß beide Eigenschaften rein qualitiativ sind und nicht quantitativ. Natürlich spielen in der Praxis noch weitere Eigenschaften bei wechselseitigem Ausschluß eine Rolle, beispielsweise daß alle Prozesse auch "etwa gleich oft" den kritischen Abschnitt betreten falls sie alle dies auch "gleich oft" wollen (also eine gewisse Art von Fairneß). Diese Bedingung kann je nach genauer Intention unterschiedlich formalisiert werden (meist als Sicherheitseigenschaft), aber es dürfte klar sein, daß jede sinnvolle Definition von wechselseitigem Ausschluß mindestens die oben beschriebene Lebendigkeit besitzen muß.

Als Daumenregel gilt: Wenn man die Verletzung einer Eigenschaft nach endlicher Zeit feststellen kann, dann ist es eine Sicherheits-, andernfalls eine Lebendigkeitseigenschaft. (Genaueres hierzu in Kapitel 3.)

2.3 Modell 0: Das asynchrone Systemmodell

Im einleitenden Abschnitt wurde bereits erwähnt, daß das Fehlen einer gemeinsamen Zeit zu den Hauptproblemen verteilter Systeme gehört. Nun ist "Zeit" ein Gegenstand mit vielen Facetten und das Fehlen einer gemeinsamen Zeit kann somit ganz unterschiedlich verstanden werden. Eine mögliche Interpretation liegt darin, daß man keine Aussage über die Ablaufgeschwindigkeiten in Teilen des Systems machen kann. Dies ist eine Grundannahme des sogenannten *asynchronen* Systemmodells, welches in diesem Abschnitt vorgestellt werden soll. Das Modell erhält die Ordnungsnummer 0, da es kein wirklich gutes Modell für fehlertolerante Systeme ist (wie zu zeigen sein wird), aber trotzdem die Basis für die in den folgenden Abschnitten vorzustellenden, verfeinerten Modelle bildet.

Das asynchrone Systemmodell kann charakterisiert werden durch die folgende Menge von Aussagen:

- Das System wird modelliert durch eine Menge von Prozessen, die durch ein Kommunikationsnetzwerk verbunden sind.
- Üblicherweise wird angenommen, daß das System *vollständig verbunden* ist, d.h. jeder Prozeß kann direkt Nachrichten an jeden anderen Prozeß senden.
- Kommunikation ist Punkt-zu-Punkt (*point-to-point*) durch zwei Anweisungen namens *send* und *receive*, d.h. man kann in *einem* Kommunikationsvorgang nur *eine* Nachricht an *einen* anderen Prozeß schicken (d.h. von Systemseite wird kein *broadcast* unterstützt).
- Es gibt keine feste Zustellungsreihenfolge auf verschickten Nachrichten, insbesondere kein *FIFO* (*first-in-first-out*).
- *Receive* und *send* sind verschiedenartige, atomare Operationen.
- Es gibt keine feste obere Schranke auf den Nachrichtenlaufzeiten.
- Der Unterschied in den Geschwindigkeiten, mit denen einzelne Prozesse ihre lokalen Algorithmen ausführen, ist unbeschränkt.

Das Attribut der Asynchronität bezieht das Modell hauptsächlich durch die letzten zwei Punkte. Nachrichten können *beliebig lange* unterwegs sein! Das bedeutet

folgendes: Eine Nachricht, die zu einem Zeitpunkt t einer fiktiven Globalzeit mittels *send* verschickt worden ist, wird garantiert am Ziel mittels *receive* empfangen, aber man kann keine Zeitschranke Δ angeben, so daß der Empfang immer vor dem Zeitpunkt $t + \Delta$ stattfindet. Auf der anderen Seite können einzelne Prozesse auch *beliebig langsam* werden. Das bedeutet, in der Zeit, die ein Prozeß benötigt, um einen Schritt seines lokalen Algorithmus auszuführen, kann ein anderer Prozeß beliebig (aber endlich) viele Schritte seines lokalen Algorithmus ausführen. Manchmal wird dieses Systemmodell auch als "zeitfrei" (*time-free*) bezeichnet [45].

Vorteile des asynchronen Systemmodells. Das asynchrone Systemmodell zeichnet sich durch seine Einfachheit aus, d.h. es werden so wenig Annahmen wie möglich gemacht. Das bedeutet, daß Systeme, in denen zusätzliche Annahmen (z.B. über maximale Nachrichtenlaufzeiten) gemacht werden, im Grunde immer das asynchrone Modell mitbeschreiben. Algorithmen, die für das asynchrone Modell entworfen worden sind und darin funktionieren, werden darum auch in jedem anderen Modell funktionieren, welches das asynchrone Modell einschließt. Auf der anderen Seite treten leicht Probleme auf, wenn Algorithmen, deren Korrektheit auf der Einhaltung von bestimmten Zeitschranken beruht, in Umgebungen ablaufen, in denen diese Zeitschranken auch nur kurzzeitig verletzt werden [162]. All dies hat das asynchrone Systemmodell zu einem sehr populären Modell für verteilte Systeme gemacht.

Zusätzlich zu den theoretischen Vorteilen ist das asynchrone Systemmodell auch von der praktischen Seite her interessant. In heutigen Weitverkehrsnetzen wie dem Internet ist es sehr schwierig (oder gar unmöglich), garantierte obere Schranken beispielsweise für Nachrichtenlaufzeiten anzugeben. Netzwerkpartitionen oder "Staus auf der Datenautobahn" können dazu führen, daß Sender oder Empfänger einer Nachricht zeitweise nicht erreichbar sind und sich die Nachrichtenzustellung manchmal über Tage hinweg verzögert [81, 125]. Andererseits werden beispielsweise auf populären Internet-Servern häufig Überlastungssituationen beobachtet, die dazu führen, daß Antworten trotz freier Datenkanäle länger als erwartet unterwegs sind. (Interessanterweise tragen gerade Mechanismen zur Herstellung von Fehlertoleranz zu diesem Phänomen bei, zum Beispiel das wiederholte Senden oder Zwischenspeichern einer Nachricht, wenn der Empfänger durch eine Netzwerkpartition temporär nicht erreicht werden kann.) Es ist also auch von einer praktischen Seite her sinnvoll, möglichst wenig (oder besser keine) Synchronitätsannahmen in das Systemmodell einzubetten [20, 35, 86, 118].

2.4 Fehler, Fehlertoleranzprobleme und “FLP”

Normalerweise wird im asynchronen Systemmodell angenommen, daß alle Netzwerkkomponenten absolut fehlerfrei arbeiten. Im Zusammenhang mit Fehlertoleranzuntersuchungen wird darum das asynchrone Modell kombiniert mit Annahmen darüber, wie und wie oft die einzelnen Teile des Systems ausfallen können. Je “schlimmer” die angenommenen Fehler, desto aufwendiger wird naturgemäß der Mechanismus sein, um diese Fehler zu tolerieren. Andererseits wird jeder (noch so gute) Fehlertoleranzmechanismus unbrauchbar, wenn die Fehlerannahme nur hinreichend “schlimm” ist und sich auch auf das Funktionieren des Fehlertoleranzmechanismus selbst bezieht. In diesem Abschnitt wird zunächst auf das Problem der Fehlermodellierung eingegangen. Anschließend wird gezeigt, daß schon unter sehr schwachen Fehlerannahmen scheinbar einfach aussehende Probleme im asynchronen Systemmodell nicht lösbar sind. All dies soll verdeutlichen, daß das asynchrone Systemmodell für Fehlertoleranzbetrachtungen nicht in jeder Hinsicht optimal ist.

2.4.1 Fehlermodelle

Eine *Fehlerannahme* (*fault assumption*) beschreibt in präziser Art und Weise das mögliche Fehlverhalten einzelner Komponenten eines Rechnersystems. Jede Form von Fehlertoleranzbetrachtung, die einen Fehlertoleranzmechanismus zum Ziel hat, benötigt als Grundlage eine Fehlerannahme. Zum Beispiel wird oft angenommen, daß maximal einer von zwei Rechnerknoten abstürzt oder daß Fehler in verschiedenen Rechnerknoten unabhängig voneinander geschehen. Will man Fehlertoleranzeigenschaften von Rechnersystemen formal nachweisen, muß die Fehlerannahme im Rahmen des Systemmodells formalisiert werden. Man spricht dann statt von Fehlerannahme auch von einem *Fehlermodell* (*fault model*).

Es ist überraschend, daß sich so unvorhersagbare und unschöne Ereignisse wie Fehler recht einfach formalisieren lassen. Der Grundgedanke der dafür verwendeten Methode basiert auf der folgenden Idee: Systeme ändern ihren Zustand aufgrund von diskreten Ereignissen aus zwei unterschiedlichen Klassen. Die erste Klasse entsteht bei der Ausführung von programmierten, d.h. “erwarteten” Anweisungen. Die zweite Klasse besteht aus “unerwarteten” Fehlereignissen. Man kann also Fehlverhalten auch in Form eines Programmes ausdrücken [42,54]. Beispielsweise ist es relativ einfach, ein Programm zu modellieren, welches statt dem richtigen Ergebnis ein falsches Ergebnis berechnet: Man nimmt das originale (korrekte) Programm und ändert darin ein paar Zeilen ab. Diese Modellierungsart funktioniert, da man offensichtlich einen Rechner, der beispielsweise einen Hardwarefehler hat, von außen nicht unterscheiden kann von einem Rechner, der diesen Hardwarefehler “softwaremäßig” simuliert.

Natürlich wird man diese Fehler bei der Softwareentwicklung von vornherein nicht in das Programm integrieren. Die Fehlermodellierung in Form von zusätzlichen oder veränderten Programmaktionen ist eher ein gedankliches Instrument, um über fehlerbehaftete Computersysteme zu argumentieren. Allerdings ist der "Einbau" von Fehlern auch in echten Softwaresystemen ein bekanntes Instrument für den Test von Computersystemen, solange die Fehler durch automatisierte Transformationen in den Programmtext eingebracht werden. Dies ist beispielsweise die Grundlage von software-implementierter Fehlerinjektion [58, 95].

Bekannte Fehlermodelle. Obwohl es teilweise Unstimmigkeiten bei der genauen Bedeutung der einzelnen Fehlerannahmen in der Literatur gibt, werden nun beispielhaft einige populäre Varianten aufgezählt und jeweils die Bedeutung festgelegt, die im weiteren Verlauf der Arbeit verwendet wird.

- *Fail-stop*: Das *fail-stop*-Fehlermodell ist sehr einfach. Die einzigen Fehler, die auftreten können, sind sogenannte *Anhalteausfälle* [102], die jedoch von anderen Prozessen leicht erkannt werden können. Salopp gesagt bedeutet *fail-stop*, daß ausgefallene Prozesse mit "Ich bin abgestürzt" antworten, wenn man ihnen eine Nachricht schickt. Oft wird dieses Fehlermodell auch als *fail-silent* bezeichnet. Der Begriff *fail-stop* geht auf einen Artikel von Schlichting und Schneider [160] zurück, in dem Methoden präsentiert werden, wie man diese Fehlerannahme auf Prozeßebene realisieren kann, selbst wenn auf niedrigerer Ebene "schlimmere" Fehler auftreten können.

- *Crash*: Das *crash*-Fehlermodell ist wie *fail-stop* mit dem Unterschied, daß fehlerhafte Prozesse hier nicht mehr antworten können. Bei *crash* muß man durch das Ausbleiben von Nachrichten auf den Zustand eines entfernten Prozesses schließen. Gerade hier liegt die Schwierigkeit des *crash*-Fehlermodells verborgen, von der später noch die Rede sein wird.

- *General omission*: Das Fehlermodell *general omission* schließt das Verhalten von *crash* mit ein, bezieht sich aber zusätzlich noch auf das Verhalten der Kommunikationskanäle. Ein Kommunikationskanal kann beliebige Nachrichten einfach "verschlucken" [89–91, 147].

- *Crash-recovery*: Im *crash-recovery*-Fehlermodell dürfen Prozesse wie im *crash*-Fehlermodell abstürzen, jedoch beginnen sie nach einiger Zeit wieder aus einem definierten Zustand heraus zu arbeiten. In diesem Fehlermodell wird weiter unterschieden, ob und wenn ja wieviel *stabilen Speicher* die Prozesse besitzen, d.h. welchen Teil ihrer Zustandsinformationen sie über einen Prozeßabsturz hinüberretten können [6, 27, 143].

- *Byzantine*: Im byzantinischen Fehlermodell geht man davon aus, daß sich einzelne Prozesse beliebig, sogar böswillig verhalten können. In der ur-

sprünglichen Definition [110] gibt es zwar die Möglichkeit zu erkennen, ob ein Prozeß (auch ein fehlerhafter) eine Nachricht gesendet hat oder nicht, jedoch wird in letzter Zeit das byzantinische Fehlermodell ebenfalls als Erweiterung von *crash* angesehen [91]. Es gibt mehrere Varianten des byzantinischen Fehlermodells: Beispielsweise wird häufig unterschieden, ob einzelne Prozesse digitale Signaturen fälschen können oder nicht [28]. Eine etwas unüblichere Klassifikation sind sogenannte *(nicht-)kooperative* byzantinische Fehler [56, 57, 134], d.h. eine Unterscheidung, ob fehlerbehaftete Rechnerknoten sich zielgerichtet "verschwören" können oder nicht.

Lokale und globale Fehlerannahme. Die eben beschriebenen Beispiele lassen sich alle als Programmtransformation auf Prozeßebene formalisieren [72]. Sie stellen aber in dieser Form nur einen Teil der Fehlerannahme dar, die sogenannte *lokale Fehlerannahme*. Um nützlich zu sein, muß man noch zusätzliche Bedingungen spezifizieren, die für alle Prozesse gelten: die *globale Fehlerannahme*. Während die lokale Fehlerannahme die Entfaltungsmöglichkeiten fehlerhafter Prozesse erweitert, schränkt die globale Fehlerannahme diese wieder ein. Fehler verlassen somit nicht einen bestimmten, vorher abgesteckten *Fehlerbereich* [55, 58]. Beispielsweise besagt die globale Fehlerannahme von *crash*, daß maximal t Prozesse abstürzen dürfen. Ähnliche Maximalgrenzen gelten in analoger Form auch bei den anderen Fehlermodellen. Prozesse, die von der Fehlerannahme betroffen werden (also im *crash*-Fehlermodell nach endlicher Zeit abstürzen), nennt man *fehlerhaft*. Alle nicht-fehlerhaften Prozesse nennt man *korrekt*.

Vergleichbarkeit der Fehlermodelle. Aufbauend auf der Formalisierung als Programmtransformation kann man die einzelnen Fehlerannahmen bezüglich ihrer "Stärke" vergleichen. Eine Möglichkeit, dies zu tun, basiert auf der Darstellung von Programmeigenschaften als Mengen und der Teilmengenrelation. Angenommen, F_1 und F_2 bezeichnen zwei Fehlerannahmen, A ist ein beliebiges Programm und $F_1(A)$ und $F_2(A)$ bezeichnen das Programm A, in dem die Fehler aus F_1 bzw. F_2 auftreten können. Ein Fehlermodell F_1 ist "schlimmer" als ein Fehlermodell F_2, falls $F_1(A)$ mehr Abläufe erlaubt als $F_2(A)$. Zu beachten ist, daß man sich bei dieser Form des Vergleiches immer auf ein konkretes Programm beziehen muß (dies wird beispielsweise von Hadzilacos und Toueg [91] nicht expliziert) [72].

Fehlererfassung. Vor der Konstruktion eines Fehlertoleranzverfahrens muß zunächst die Fehlerannahme fixiert werden. Die Wahl einer adäquaten Fehlerannahme ist situationsabhängig. Bei einfachen betriebsinternen Systemen sind *crash* oder *fail-stop* normalerweise vollkommen ausreichend, während bei sicher-

heitsrelevanten Anwendungen wie Steuerungssystemen von Kriegsschiffen [167] das byzantinische Fehlermodell bevorzugt wird. Wie oben erläutert, gibt es immer Szenarien, die in der Wirkung "schlimmer" sind als das angenommene Fehlermodell, d.h. kein sinnvolles Fehlermodell deckt alle Fehler ab, die in der Realität auftreten können, da es ansonsten nicht tolerierbar wäre. Jedoch dürfte deutlich geworden sein, daß das byzantinische Fehlermodell einen größeren Prozentsatz an realen Fehlersituationen beschreibt als beispielsweise *fail-stop*. Diesen Prozentsatz bezeichnet man üblicherweise als *Fehlererfassung* (*fault coverage*) [58, 150].

Der Grad der Fehlererfassung ist eine Komponente in statistischen Verläßlichkeitsbetrachtungen, d.h. in Situationen, wo beispielsweise die prozentuale Verfügbarkeit pro Jahr berechnet werden muß. Da jedoch die Menge der in einer konkreten Anwendungssituation auftretenden Fehler normalerweise unbekannt ist, müssen in der Praxis bei der Bezifferung der Fehlererfassung immer Erfahrungswerte herhalten. Trotz einzelner empirischer Untersuchungen [32, 106] ist Erfahrung auch unabdingbar, wenn es um die Wahl einer guten, d.h. realistischen Fehlerannahme für ein bestimmtes Anwendungsgebiet geht. Die Fixierung einer Fehlerannahme erlaubt es, von derartigen Problemen zu abstrahieren und Aussagen zu treffen von der Art "unter Fehlerannahme x erfüllt Programm y die Eigenschaft z", d.h. Aussagen, die ohne Wahrscheinlichkeiten auskommen. Das sind genau die Aussagen, die bei der Verifikation von Fehlertoleranzalgorithmen eine Rolle spielen und um die es in dieser Arbeit geht.

2.4.2 Übereinstimmungsprobleme

Zu den wichtigsten Klassen von Problemen in der Fehlertoleranz gehören die sogenannten *Übereinstimmungsprobleme* [28] (*agreement problems*). Diese Probleme tauchen in vielfältigen Variationen auf. Zur Motivation folgen einige Beispiele:

- Replizierte Datenbanken [28, S. 9f]: Um gegen den Ausfall einzelner Rechner geschützt zu sein, werden die Datenbestände einer Bank auf mehreren, geographisch verteilten Rechnern vorgehalten. Transaktionen auf diesem Datenbestand sollen natürlich auf allen Rechnern (zumindest den fehlerfreien) gleich ablaufen. Dafür ist es notwendig, daß sich alle beteiligten Rechner bezüglich des Ausgangs der Transaktion koordinieren und sich auf einen einheitlichen Ausgang einigen.

- Redundante Sensoren [126, S. 81]: In einem kommerziellen Linienflugzeug wird der Sensor für die Flughöhe mehrmals installiert, um sich gegen fehlerhafte Messdaten einzelner Höhenmesser zu schützen. Bevor ein Wert an der Konsole angezeigt werden kann, müssen die beteiligten Prozesse in Echtzeit einen gemeinsamen "echten" Wert aus der Menge der gemessenen Werte ermitteln.

- Fehlerdiagnose [126, S. 81]: Im Steuerungssystem eines Atomkraftwerks arbeiten mehrere an verschiedenen Orten aufgestellten Rechnersysteme parallel, um gegen den Ausfall eines einzelnen Systems geschützt zu sein. Ein fehlerhaftes System soll automatisch erkannt und abgeschaltet werden. Hierbei müssen die beteiligten Rechner zu einer einheitlichen Entscheidung gelangen, welches System als fehlerhaft und welches als korrekt bezeichnet wird.

- Versteigerungen im Internet [28, S. 10]: In den letzten Jahren sind öffentliche Versteigerungen im Internet populär geworden. Dabei ist es jedoch notwendig, daß alle Teilnehmer sowohl dieselbe Sicht auf die versteigerten Objekte haben, als auch dieselben aktuellen Gebote sehen. Die beteiligten Endgeräte (oder *browser*) des Auktionssystems müssen sich demnach bezüglich der Sicht auf den momentanen Stand der Versteigerung einigen und allen beteiligten Bietern diesen Stand möglichst gleichzeitig anzeigen.

- Fairer Austausch [144,179]: Inzwischen gibt es elektronische Zahlungssysteme, die es erlauben, mittels *elektronischer Münzen* Waren direkt im Internet zu bezahlen. Bei einem solchen Verkaufsvorgang ist es natürlich wichtig, daß sich Verkäufer und Kunde darauf einigen, welche Eigenschaft das Produkt haben sollte und wieviel dafür zu bezahlen ist. Noch wichtiger ist, daß beide beteiligten Parteien am Ende des Austausches von Geld gegen Ware auch tatsächlich das haben, was sie erhalten wollten und niemand einen Nachteil erleidet.

- Mobile Agenten [148, 154]: Das Paradigma der *mobilen Agenten* sieht vor, daß autonome Programmeinheiten (die Agenten) im Auftrag eines Benutzers von Rechner zu Rechner wandern und dort jeweils definierte Aktionen durchführen wie beispielsweise das Sammeln von Informationen. Nach einer bestimmten Zeit kehrt der Agent mit den gesammelten Daten zum Benutzer zurück, der in der Zwischenzeit keine aktive Netzwerkverbindung aufrechterhalten muß. Da die Rechner, über die der Agent wandert, ausfallen können, muß ein Fehlertoleranzverfahren dafür sorgen, daß einerseits der Agent nicht verloren geht aber andererseits auch nicht mehr als eine Instanz des Agenten im Netz existiert. Wenn ein Agent also von einem Rechner zum anderen wandert, müssen Start- und Zielrechner auch im Fehlerfall eine Übereinstimmung erzielen über den Ausgang dieses Schrittes. Dieses Problem wird oft auch als das *exactly once copy*-Problem von mobilen Agenten bezeichnet.

Nicht in allen diesen Beispielen muß exakt dasselbe Problem gelöst werden. Beispielsweise wird der Wert, auf den sich die replizierten Sensoren einigen, der "Mehrheitswert" aller Sensoren sein, während sich bei den replizierten Datenbanken die Replikate auf einen Abbruch der Transaktion einigen müssen, sobald auch nur ein einziges Replikat dies wünscht. Es gibt Bücher, die in großen Teilen

oder ausschließlich verschiedene Variationen von Übereinstimmungsproblemen betrachten [28, 126, 127]. Aus der Fülle der bereits untersuchten Übereinstimmungsprobleme hat sich eine Variante herauskristallisiert, die den Kern vieler solcher Probleme beinhaltet und sich deshalb gut für die abstrakte Untersuchung eignet: das *consensus*-Problem.

Das *consensus*-Problem. Die Definition des *consensus-Problems* basiert auf zwei abstrakten Operationen, die *propose* und *decide* genannt werden. Beide Operationen benötigen als Aufrufparameter einen Wert aus einer endlichen Menge von Entscheidungswerten. Ruft ein Prozeß die Operation $propose(v)$ auf, heißt das, daß dieser Prozeß den Wert v zur Entscheidung *vorschlägt*. In analoger Weise heißt es, daß ein Prozess den Wert v *entscheidet*, wenn er $decide(v)$ aufruft. Ein Algorithmus, der das *consensus*-Problem löst, geht davon aus, daß alle beteiligten Prozesse jeweils einen bestimmten Wert vorschlagen, und muß dann die folgenden drei Eigenschaften erfüllen:

1. Übereinstimmung (*agreement*): Es ist nie der Fall, daß zwei Prozesse unterschiedliche Werte entscheiden.

2. Terminierung (*termination*): Jeder Prozeß entscheidet sich nach endlicher Zeit.

3. Gültigkeit (*validity*): Der entschiedene Wert muß von mindestens einem Prozeß vorgeschlagen worden sein.

Die Gültigkeits-Bedingung ist eine sogenannte "Nichttrivialitäts-Bedingung", d.h. sie wurde nur hinzugefügt, um triviale Lösungen auszuschließen (insbesondere Lösungen, in denen nicht kommuniziert wird). Bedingung 1 und 3 sind Sicherheitseigenschaften des Algorithmus, Bedingung 2 ist eine Lebendigkeitseigenschaft [38].

***Consensus* unter verschiedenen Fehlerannahmen.** Die obige Definition von *consensus* ist allgemein in dem Sinne, daß sie ohne eine spezielle Fehlerannahme formuliert worden ist. Die Problemdefinition muß in der Regel an eine gewählte Fehlerannahme angepaßt werden. Nimmt man beispielsweise eine Fehlerannahme wie *crash* hinzu und würde man weiterhin verlangen, daß *jeder* Prozeß nach endlicher Zeit entscheidet, stünde man vor einem unlösbaren Problem: Es kann nämlich immer passieren, daß ein Prozeß abstürzt, bevor er entschieden hat. Da das *crash*-Fehlermodell aber lediglich die Lebendigkeit des Algorithmus beeinflußt [73], genügt es, die Bedingung 2 in folgender Art und Weise abzuschwächen:

- Terminierung: Jeder korrekte Prozeß (d.h., jeder, der nicht abstürzt) entscheidet sich nach endlicher Zeit.

Andere Fehlertoleranzprobleme. Wie oben bereits beschrieben, ist das *consensus*-Problem ein wichtiges Problem in der Fehlertoleranz, da es den Kern vieler Übereinstimmungsprobleme bildet und Übereinstimmungsprobleme in der Fehlertoleranz sehr wichtig sind. Andere Probleme, wie zum Beispiel das des *atomic broadcast* [35], das Transaktionsproblem (*atomic commitment*) [83] oder das *election*-Problem [156] lassen sich auf *consensus* zurückführen. Es existieren aber auch Probleme, die gänzlich anders sind, beispielsweise das zuverlässige Verschicken von Nachrichten an alle Prozesse (*reliable broadcast*). Es hat sich jedoch herausgestellt, daß eine Lösung des *consensus*-Problems unter einer bestimmten Fehlerannahme normalerweise ausreicht, um fehlertolerante Anwendungen für diese Fehlerannahme zu realisieren.

2.4.3 Das FLP-Theorem

Im Jahre 1985 erschien ein Artikel, der endlich zweifelsfrei nachwies, was viele schon längere Zeit vermutet hatten: Das *consensus*-Problem ist deterministisch unter der *crash*-Fehlerannahme in asynchronen Systemem nicht lösbar [68], d.h. es existiert kein Algorithmus, der in jedem Fall die Eigenschaften aus Abschnitt 2.4.2 erfüllt. Das entsprechende Theorem wird nach seinen Autoren (Michael Fischer, Nancy Lynch und Michael Paterson) mit "FLP" bezeichnet.

Das *consensus*-Dilemma. Statt den formalen Beweis aus dem Originalartikel [68] zu wiederholen, wird anhand eines Beispiels das dem Theorem zugrundeliegende Dilemma erläutert (etwas allgemeinverständlichere Versionen des Resultates sind an anderer Stelle [49, 172, 175, 181] bereits erschienen).

Ausgangspunkt des Beispiels sind zwei Prozesse p_1 und p_2 im asynchronen Systemmodell. Die Prozesse sind durch eine bidirektionale Kommunikationsleitung verbunden, welche fehlerfrei funktioniert. Für Prozesse gilt allerdings die *crash*-Fehlerannahme, d.h. es kann jederzeit passieren, daß ein Prozeß einfach anhält. Angenommen, p_1 wartet auf eine Nachricht m von p_2, in der wichtige Informationen übermittelt werden, die p_1 zur eigenen Entscheidungsfindung braucht. Solange wie p_1 die Nachricht noch nicht erhalten hat, steht dieser Prozess vor dem Dilemma, wie lange noch auf m gewartet werden soll. Im Grunde hat der Prozeß zwei Wahlmöglichkeiten:

1. p_1 kann einfach noch länger warten, oder

2. p_1 wird irgendwann ungeduldig und nimmt an, daß m nicht mehr ankommen wird, weil p_2 vor dem Senden von m abgestürzt ist.

Welche Wahl ist die richtige? Sich generell für die erste Alternative zu entscheiden, wäre fatal, wenn p_2 tatsächlich abgestürzt ist. Dann könnte p_1 unter Umständen nie zu einer Entscheidung gelangen (wenn nämlich p_2 abstürzt bevor m abgeschickt wurde). Dies verletzt aber die geforderte Eigenschaft der Terminierung.

Wenn p_1 sich allerdings für die zweite Alternative entscheidet, funktioniert alles prächtig, solange wie die Annahme über den Absturz von p_2 richtig ist. In diesem Fall kann sich p_1 für einen beliebigen Wert entscheiden (am besten den Wert, den der Prozeß selbst vorgeschlagen hat). Man beachte allerdings, daß dasselbe Argument symmetrisch auch für p_2 formuliert werden kann, d.h. p_2 wartet auf eine Nachricht von p_1 und nimmt dann an, p_1 sei abgestürzt und entscheidet sich für den eigenen Wert. Es kann aber durchaus sein, daß beide Prozesse unterschiedliche Werte vorgeschlagen haben mit dem Resultat, daß beide Prozesse unterschiedliche Werte entscheiden. Wenn man also die zweite Alternative wählt, läuft man Gefahr, die Eigenschaft der Übereinstimmung zu verletzen.

Im asynchronen Systemmodell gibt es keine Möglichkeit allgemein zu entscheiden, welche Alternative in der gegebenen Situation die richtige ist. Das Dilemma ist unlösbar.

Lösbare Probleme unter *crash*. Es ist erwähnenswert, daß nicht alle Fehlertoleranzprobleme unter der *crash*-Fehlerannahme unlösbar sind (ein Beispiel hierfür ist das Problem des *reliable broadcast* [35, 91]). In vielen Fällen werden auch Fehlerannahmen getroffen, welche die Problematik von *crash* geschickt wegdefinieren. Beispielsweise geht die originale Annahme über byzantinische Fehler von Lamport [110] davon aus, daß man feststellen kann, ob noch eine Nachricht von einem anderen Prozeß *unterwegs* ist oder nicht. So werden Lösungen für das Problem der byzantinischen Generäle [110] (eine Erweiterung von *consensus*) im asynchronen Systemmodell wieder möglich.

Folgerungen aus FLP. Daß das so wichtige *consensus*-Problem unter einem so einfachen Fehlermodell wie *crash* unlösbar ist, hat weitreichende Folgen: Zum einen bedeutet dies, daß *consensus* auch unter allen anderen Fehlerannahmen unlösbar ist, die *crash* beinhalten (und das sind viele); zum anderen zeigt es, daß das auf den ersten Blick so gut geeignete asynchrone Systemmodell für Fehlertoleranzbetrachtungen eigentlich ungeeignet ist.

Es gibt prinzipiell zwei verschiedene Möglichkeiten für "ein Leben nach FLP" [86]: Man kann entweder das Problem abschwächen oder das Systemmodell verstärken.

Ersteres kann beispielsweise durch die Einführung von Randomisierung erreicht werden [8, 25, 43], soll hier aber nicht weiterverfolgt werden. In der Literatur hat es verschiedene Versuche gegeben, zusätzliche Annahmen in das asynchrone Systemmodell aufzunehmen, um *consensus* lösbar zu machen. Diese verschiedenen Annahmen führten im wesentlichen zu vier alternativen Systemmodellen, die im folgenden vorgestellt werden sollen.

2.5 Modell 1: Einführung expliziter Zeitschranken

2.5.1 Synchrone Systeme

Die Einführung expliziter Zeitschranken ist vielleicht die einfachste Möglichkeit, die in Abschnitt 2.4.3 beschriebenen Nachteile des asynchronen Systems zu umgehen. Die Zeitschranken beziehen sich auf die Nachrichtenlaufzeiten und den Unterschied in den Geschwindigkeiten, mit denen die Prozesse ihre lokalen Algorithmen abarbeiten. Die Werte δ und Δ dieser Schranken haben folgende Bedeutung:

- Abarbeitungsgeschwindigkeiten: In der Zeit, die ein beliebiger Prozeß braucht, um δ Schritte seines lokalen Algorithmus abzuarbeiten, führen alle anderen Prozesse mindestens einen Schritt aus.
- Nachrichtenlaufzeiten: Wenn ein Prozeß p eine Nachricht m verschickt, dann muß m nach spätestens Δ Schritten des lokalen Algorithmus von p beim Adressaten ausgeliefert werden.

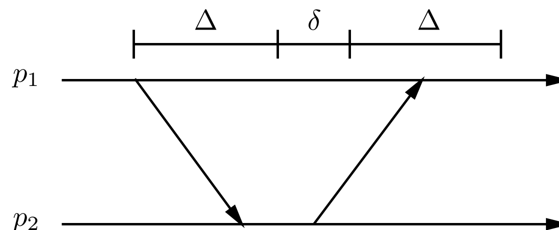


Abbildung 2.1: Wie lange muß man warten, damit der Absturz von p_2 erkannt werden kann?

Wenn die Zeitschranken δ und Δ existieren und bekannt sind, dann ist es für eine Prozeß p_1 offensichtlich einfach, den Absturz eines entfernten Prozesses p_2 zu entdecken. Man schickt dem entfernten Prozeß eine Nachricht, in der man

ihn bittet, sofort zu antworten. Wenn diese Antwort nicht innerhalb von $2\Delta + \delta$ Schritten empfangen wird, dann ist p_2 sicher abgestürzt (siehe Abbildung 2.1). Durch diese neue Informationsmöglichkeit wird das oben beschriebene *consensus*-Dilemma hinfällig. Es ist nun auf relativ einfache Art und Weise möglich, *consensus*-Algorithmen zu entwerfen, die selbst dann noch korrekt sind, wenn *alle* Prozesse abstürzen [49]. Dabei ist zu beachten, daß — solange nicht zwei Prozesse unterschiedlich entscheiden — eine Ausführung, in deren Verlauf alle Prozesse abstürzen, die geforderten Eigenschaften des *consensus*-Problems trivialerweise immer noch erfüllt.

Wenn *send* und *receive* zwei separate Anweisungen sind, dann sind beide Schranken δ und Δ für die Fehlererkennung notwendig. Dolev, Dwork und Stockmeyer [49] haben gezeigt, daß manchmal eine oder sogar beide Schranken nicht benötigt werden, wenn andere Systemannahmen getroffen werden. Beispielsweise ist *consensus* ohne die beiden Schranken lösbar, wenn man eine Nachricht in einer ununterbrechbaren Operation an alle Prozesse senden kann (*broadcast*) und wenn diese Nachrichten bei den Adressaten in der "Echtzeit"-Reihenfolge zugestellt werden, in der sie abgeschickt wurden.

2.5.2 Partiell synchrone Systeme

Selbst wenn die in Abschnitt 2.5.1 geforderten Zeitschranken existieren, ist es in der Praxis oft schwierig, sie zu benennen. Man möchte also gerne Systemmodelle haben, die etwas schwächere Annahmen über die gegebenen Zeitschranken machen, aber trotzdem noch stark genug sind, um *consensus* zu lösen. Dies hat zum Konzept der *partiell synchronen Systeme* (*partially synchronous systems*) geführt [53].

Es gibt zwei Varianten von partieller Synchronität, die hier anhand der Zeitschranke Δ auf den Nachrichtenlaufzeiten verdeutlicht werden sollen:

1. Die Schranke Δ existiert; sie ist aber nicht bekannt.
2. Die Schranke Δ ist bekannt, aber sie gilt nicht sofort sondern erst nach einer unbekannten endlichen Zeit.

Wenn eine dieser beiden Bedingungen für die Zeitschranke Δ gilt, spricht man von *partiell synchroner Kommunikation* (*partially synchronous communication*). Man kann die Definition auch in analoger Weise auf die Zeitschranke δ anwenden und spricht dann von *partiell synchronen Prozessen* (*partially synchronous processes*).

Beide Varianten von partieller Synchronität spiegeln durchaus praktische Probleme wieder. Für gewöhnlich nimmt man schon an, daß beispielsweise eine maximale Nachrichtenlaufzeit existiert, doch sie tatsächlich anzugeben, ist oft nicht

möglich (so groß man sie auch wählen mag). Dies motiviert die erste Form partieller Synchronität. Die zweite Variante wird dadurch motiviert, daß Systeme in der Praxis normalerweise einem Wechsel zwischen kurzen "schlechten" Zeiträumen und langen "guten" Zeiten unterliegen [45]. Während der guten Zeiten ist das System stabil und die angenommenen Echtzeitschranken Δ und δ gelten uneingeschränkt. In den schlechten Zeiten ist das System instabil und die Zeitschranken gelten nicht (beispielsweise können Computer infolge einer Überlastsituation "zu langsam" sein). Da die guten Zeiten normalerweise lang genug sind, um einen einmal gestarteten Algorithmus zuende auszuführen, muß man nur noch den Fall berücksichtigen, daß ein Algorithmus während oder kurz vor dem Beginn einer instabilen Phase gestartet wird.

Interessanterweise ist *consensus* lösbar unter beiden Arten der partiellen Synchronität, selbst wenn sie sowohl die Kommunikation als auch die Prozesse betreffen. Dies soll im folgenden kurz erläutert werden.

Nehmen wir beispielsweise partiell synchrone Kommunikation in der ersten Form an, d.h. es existiert eine obere Schranke Δ auf den Nachrichtenlaufzeiten, aber sie ist nicht bekannt. Die Idee ist nun, einfach eine erste Approximation Δ' von Δ anzunehmen und einfach mittels des in Abschnitt 2.5.1 vorgestellten Verfahrens Abstürze eines entfernten Prozesses zu erkennen. Falls die Zeitschranke Δ' zu klein gewählt worden ist, kann dies allerdings dazu führen, daß man einen entfernten Prozeß fälschlicherweise als abgestürzt annimmt und im Rahmen des *consensus*-Dilemmas die falsche Entscheidung trifft; eine Verletzung der Sicherheitseigenschaft von *consensus* wäre die Folge. Es ist jedoch möglich, *consensus*-Algorithmen zu entwerfen, bei denen derartige falsche Fehlerentdeckungen nicht die Sicherheit gefährden, sondern lediglich den Algorithmus bei seiner Entscheidungsfindung verzögern. Dies soll an einem Beispiel verdeutlicht werden.

Ein *consensus*-Algorithmus, der nie die Sicherheit verletzt. Betrachten wir ein asynchrones System mit partiell synchroner Kommunikation. In dem System existieren drei Prozesse p_1, p_2 und p_3, von denen maximal einer abstürzen kann. Der Algorithmus läuft ab in mehreren *Runden*, von denen jede in vier *Phasen* gegliedert ist. Jeder Prozeß besitzt eine lokale Variable *estimate*, in der er den aktuellen "Schätzwert" der zu treffenden Entscheidung speichert. Die Variable *estimate* wird initialisiert mit dem Wert, den der jeweilige Prozeß vorgeschlagen hat.

Damit die Nachrichten aus verschiedenen Runden unterschieden werden können, wird jede Nachricht mit der Nummer der aktuellen Runde gekennzeichnet, in der sie abgeschickt wurde. Zusätzlich gibt es in jeder Runde unter den Prozessen einen sogenannten *Koordinator*. In der ersten Runde ist dies Prozeß p_1, in der zweiten Runde p_2, und so geht es reihum weiter. In den vier Phasen der ersten Runde passiert das Folgende:

```
1  Process p_i:

2    variables:
3      estimate = ⟨proposed value⟩
4      round = 1                                          {* current round number *}
5      coord = 1                                          {* current coordinator *}
6      update = 1              {* last round number in which estimate was updated *}

7    algorithm:
8      while ⟨not yet decided⟩
9                                                                   {* phase 1: *}
10       send (estimate, round, update) to p_coord
11                                                                  {* phase 2: *}
12       if i = coord then
13         receive (e, r, u) from ⟨a non-coordinator⟩
14         estimate := ⟨either estimate or e depending on update and u⟩
15         send (estimate, round) to all
16                                                                  {* phase 3: *}
17       wait until ⟨(e, r) is received from p_coord⟩ or ⟨p_coord is suspected⟩
18       if ⟨(e, r) received⟩ then
19         estimate := e
20         update := u
21         send ack to p_coord
22       else
23         send nack to p_coord
24                                                                  {* phase 4: *}
25       if i = coord then
26         wait for ⟨at least one answer from a non-coordinator⟩
27         if ⟨received at least one ack⟩ then
28           ⟨decide on estimate, tell the others and exit⟩
29                                                      {* switch to next round: *}
30       round := round + 1
31       coord := (round mod 3) + 1
32     end
```

Abbildung 2.2: Ein *consensus*-Algorithmus, der niemals die Sicherheit verletzt.

- In Phase 1 schickt jeder Prozeß seinen aktuellen Schätzwert an den Koordinator p_1.
- In Phase 2 wartet p_1 auf den Schätzwert von mindestens einem weiteren Prozeß (d.h. von p_2 oder p_3). Prozeß p_1 wählt nun unter den eingegangenen Werten (und seinem eigenen) einen neuen Schätzwert, den er in seiner lokalen *estimate*-Variable speichert und an die anderen Prozesse p_2 und p_3 verschickt. (Zu beachten ist folgendes: Wenn p_1 bis jetzt noch nicht abgestürzt ist, wird diese Phase garantiert terminieren.)
- In Phase 3 speichern alle Prozesse, die den neuen Schätzwert vom Koordinator erhalten, diesen neuen Wert in ihrer eigenen *estimate*-Variable und schicken eine positive Bestätigung an den Koordinator zurück.
- In Phase 4 wartet der Koordinator p_1 auf die Bestätigungen der anderen Prozesse. Erhält er mindestens eine positive Bestätigung von p_2 oder p_3, kann er sich für den aktuellen Wert in *estimate* entscheiden und den anderen Prozessen mitteilen, daß sie dasselbe tun sollen.

Für das Verständnis des Algorithmus ist folgende Beobachtung entscheidend: Der einzige Punkt, an dem der Absturz eines Prozesses das Fortschreiten des Algorithmus hindern kann, ist in Phase 3. In dieser Phase warten alle Prozesse, die gerade nicht Koordinator sind, auf eine Nachricht des Koordinators. Wenn eben dieser Koordinator jedoch vor dem Abschicken dieser Nachricht abstürzt, könnten die Prozesse unendlich lange warten und somit die Terminierungseigenschaft von *consensus* verletzen.

An dieser Stelle ist es also sinnvoll, *zeitüberwacht* zu warten und bei der Überschreitung einer gegeben Zeit einfach anzunehmen, der Koordinator sei ausgefallen (die Zeitschranke kann beispielsweise auf der Vermutung Δ' basieren). In diesem Fall schickt ein Prozeß trotzdem eine *negative* Bestätigung an den möglicherweise abgestürzten Koordinator und schaltet in die nächste Runde des Algorithmus weiter. Es ist also nun möglich, daß ein Koordinator in Phase 4 eine oder zwei negative Bestätigungen empfängt. In diesem Fall verschiebt er die eigene Entscheidung und schaltet ebenfalls um zur nächsten Runde. In der nächsten Runde beginnt alles von vorne mit einem neuen Koordinator.

Erreichen der Sicherheit. Warum wird hier die Sicherheit des Algorithmus nie gefährdet? Damit dies eintritt, müssten zwei Prozesse unterschiedliche Werte entscheiden. Das kann aber nur passieren, wenn zur Entscheidungsfindung des Koordinators Schätzwerte aus verschiedenen Runden vorliegen. Angenommen, der Koordinator p_1 entscheidet sich in Phase 4 der ersten Runde für den Wert v und stürzt ab, bevor er diese Mitteilung an die anderen Prozesse abschicken kann. Angenommen, p_2 hat in der ersten Runde den Koordinator verdächtigt

und eine negative Bestätigung an p_1 geschickt, ohne den eigenen Schätzwert zu aktualisieren. Zu Beginn der zweiten Runde sind also nur p_2 und p_3 noch am Leben. Jetzt spielt p_2 den Koordinator, erhält als Entscheidungsgrundlage den eigenen (initialen) Schätzwert und den Schätzwert v von p_3. Prozeß p_2 darf sich jetzt offensichtlich *nicht* für seinen eigenen Schätzwert entscheiden, da sonst die Sicherheit verletzt werden könnte. Man muß in diesem Falle den anderen Schätzwert benutzen.

Allgemein darf ein Koordinator immer nur die neusten Schätzwerte zur Entscheidungsfindung benutzen, d.h. jeder Prozeß sendet neben dem Schätzwert auch immer noch die Rundennummer, in der dieser Schätzwert angenommen wurde. Wenn eine Entscheidung für einen Wert v gefallen ist, dann hat mindestens die Hälfte der Prozesse den Wert v als Schätzwert bereits angenommen. Wenn dies einmal geschehen ist, kann kein anderer Wert als v der Entscheidungswert sein. Die Sicherheit bleibt also auch bei fehlerhaften Absturzvermutungen (und damit auch fehlerhaften Schätzungen von Δ) in jedem Fall erhalten. Nur dauert es manchmal länger, bis der Algorithmus terminiert.

Erreichen der Lebendigkeit. Um die Lebendigkeit des Algorithmus zu erreichen, muß man nur garantieren, daß nach endlicher Zeit keine falschen Absturzmeldungen verursacht werden. Im Falle partiell synchroner Kommunikation basieren solche Falschmeldungen auf einer zu klein gewählten Schranke Δ', d.h. Falschmeldungen treten nicht mehr auf sobald $\Delta' \geq \Delta$. Man kann also in jeder Runde, in der es zu keiner Entscheidung kommt, die Vermutung Δ' um einen konstanten Wert erhöhen. Dies führt dazu, daß nach endlicher Zeit die Bedingung $\Delta' \geq \Delta$ wahr wird, daß also keine Falschmeldungen mehr verursacht werden und somit der Algorithmus terminiert.

In der zweiten Art von partiell synchroner Kommunikation, in der die Schranke Δ erst nach endlicher Zeit gilt, ist das Verfahren noch einfacher. In diesem Fall kann man den Algorithmus einfach starten, als gelte Δ bereits (Falschmeldungen können die Sicherheit ja nicht gefährden). Nach endlicher Zeit wird Δ aber gelten und somit der Algorithmus terminieren.

Die Bedingung $t < n/2$. In der oben skizzierten Argumentation ist ein Punkt noch relativ wichtig: Es wird angenommen, daß es immer eine Mehrheit an funktionstüchtigen Prozessen im System gibt. Dies ist Teil der Fehlerannahme, d.h. es gilt die *crash*-Fehlerannahme mit $t < n/2$ (wobei t die maximale Anzahl an fehlerhaften Prozessen angibt). In einem gewissen Sinne ist dies der Preis, den man dafür bezahlen muß, um mit fehlerhaften Absturzmeldungen fertig zu werden.

Umgekehrt ist es relativ einfach zu zeigen, daß *consensus* wieder unlösbar wird,

wenn man $t \geq n/2$ zuläßt. Eine Menge von n Prozessen kann sich nämlich "virtuell" partitionieren, d.h. es kann zwei Prozeßgruppen geben, welche die Mitglieder der jeweils anderen Gruppe für tot erachten. In diesen getrennten Gruppen kann es nun zu unterschiedlichen Entscheidungen und damit zu einer Verletzung der Sicherheit kommen. Man muß darum fordern, daß eine solche Partition immer die Mehrheit an Prozessen enthält. So kann es nicht vorkommen, daß zwei Partitionen unterschiedlich entscheiden, da es keine zwei Mehrheitspartitionen geben kann.

2.6 Modell 2: Benutzung von Fehlerdetektoren

In den (partiell) synchronen Systemmodellen konnte man die gegebenen Zeitschranken dafür benutzen, um den Absturz eines Prozesses festzustellen oder wenigstens zu vermuten. Mit Hilfe dieser Informationen war es dann möglich, *consensus*-Algorithmen für das gegebene Systemmodell zu entwerfen. Softwaretechniker werden diesen Ansatz jedoch ungenügend finden, da hier zwei Bereiche vermischt werden, die zur Erhöhung der Modularität der Lösung eigentlich auseinandergehalten werden sollten:

1. die Schnittstelle, d.h. die abstrakte Funktionalität, Prozeßabstürze erkennen zu können, und

2. die Implementierung, d.h. die Art und Weise, wie die Funktionalität unter Zuhilfenahme der angenommenen Zeitschranken implementiert wird.

Diese Beobachtung zeigt, daß es offensichtlich möglich ist, *consensus*-Algorithmen in Systemen ohne spezielle Synchronitätsannahmen zu entwickeln, solange man die Möglichkeit hat, Prozeßabstürze sicher zu erkennen. Diese Idee ist der Kern des Konzeptes der *Fehlerdetektoren* (*failure detectors*) [35], auf das in diesem Abschnitt eingegangen werden soll.

2.6.1 Perfekte Fehlerdetektoren

Fehlerdetektoren sind Programmmodule, mit deren Hilfe einzelne Prozesse Informationen über den "Lebenszustand" entfernter Prozesse gewinnen können. Fehlerdetektoren sind in einer ersten Näherung "Zustands-Orakel", deren interne Realisierung der Benutzer nicht kennt.

Schnittstelle innerhalb eines Prozesses. Zur Einführung in das Konzept der Fehlerdetektoren betrachten wir das Szenario, welches in Abbildung 2.3 dar-

gestellt ist. Prozeß p_1 besitzt einen Fehlerdetektor, mit dem er den Funktionszustand von Prozeß p_2 abfragen kann. Bei jeder Anfrage liefert der Fehlerdetektor entweder den Wert *up* oder *down*, je nachdem ob p_2 noch lebt oder ob der Prozeß abgestürzt ist. Um nützlich zu sein, sollte der Fehlerdetektor die folgenden beiden Eigenschaften besitzen:

- Es ist nie der Fall, daß der Fehlerdetektor den Wert *down* ausgibt, ohne daß p_2 zu diesem Zeitpunkt tatsächlich abgestürzt ist.
- Wenn p_2 abstürzt, dann wird nach endlicher Zeit der Fehlerdetektor nur noch den Wert *down* zurückgeben.

Die erste Eigenschaft ist eine Sicherheitseigenschaft; sie bewahrt den Fehlerdetektor vor falschen Alarmen. Ein trivialer Fehlerdetektor, der immer und ausschließlich die Sicherheit erfüllt, würde darum aus Angst vor falschen Verdächtigungen niemals einen anderen Prozeß verdächtigen. Darum muß zusätzlich die zweite (Lebendigkeits-)Eigenschaft gefordert werden. Sie fordert, daß ein eventueller Absturz des entfernten Prozesses auch nach endlicher Zeit erkannt werden muß. Die beiden Eigenschaften des Fehlerdetektors zeigen, daß es sich bei ihm um eine Instanz des *Problems von der Erkennung von Prädikaten* (*predicate detection problem*) [36] handelt. Andere Beispielinstanzen sind das Problem der Terminierungserkennung [48, 135] oder das der Erkennung von Verklemmungen in einem verteilten System.

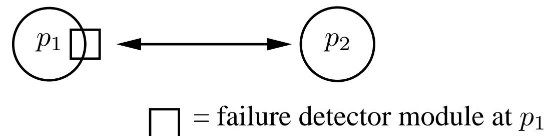


Abbildung 2.3: Ein einfaches Fehlererkennungsszenario.

Man kann in einfacher Art und Weise die Definition des Fehlerdetektors aus Abbildung 2.3 auf zwei Prozesse erweitern. Man fordert einfach, daß der Fehlerdetektor von p_2 in analoger Weise die Fehler von p_1 erkennt. Gemäß der Terminologie von Chandra und Toueg [35] heißen Fehlerdetektoren, die lokal auf den einzelnen Prozessen residieren, *lokale Fehlerdetektoren* oder *Fehlerdetektormodule* (*failure detector modules*), und die Gesamtheit von lokalen Fehlerdetektoren sind der eigentliche Fehlerdetektor.

Wie im Beispiel mit zwei Prozessen kann man die Definition eines Fehlerdetektors auch leicht auf ein Gesamtsystem mit n Prozessen erweitern. Dabei ist jedoch zu beachten, daß jeder lokale Fehlerdetektor für die Zustandsbeobachtung von *allen* anderen Prozessen verantwortlich ist. Eine Anfrage an einen lokalen Fehlerdetektor liefert also eine Liste von *up*/*down*-Werten, einen für jeden entfernten Prozeß.

Im folgenden wird darum vereinfachend angenommen, daß die Schnittstelle eines lokalen Fehlerdetektors aus einem einfachen Booleschen Feld $up[1..n]$ mit n Einträgen besteht. Ein Wert von $up[j] = false$ im Prozeß p_i bedeutet, daß der lokale Fehlerdetektor von p_i vermutet, p_j sei abgestürzt. Manchmal sagt man in diesem Fall auch, p_i *verdächtigt* p_j (p_i *suspects* p_j). Wenn eine einmal gemachte Verdächtigung nicht widerrufen wird, sagt man auch, p_i *verdächtigt dauerhaft* p_j (p_i *permanently suspects* p_j).

In einem System mit n Prozessen besteht die Sicherheitseigenschaft des Fehlerdetektors einfach aus den einzelnen Sicherheitseigenschaften der lokalen Fehlerdetekoren. Dies kann wie folgt formuliert werden:

- Für alle Prozesse p: Für alle Prozesse q:
  Es ist nie der Fall, daß p den Prozeß q verdächtigt, ohne daß q tatsächlich abgestürzt ist.

Analog kann die Lebendigkeitseigenschaft formuliert werden als:

- Für alle korrekten Prozesse p: Für alle Prozesse q:
  Falls q abstürzt, dann wird p den Prozeß q nach endlicher Zeit dauerhaft verdächtigen.

Die Einschränkung der Lebendigkeitseigenschaft auf alle korrekten Prozesse erfolgt analog zu der Adaption der Terminierungseigenschaft von *consensus* in Abschnitt 2.4.2: Man kann nicht von einem abstürzenden Prozeß erwarten, daß er nach endlicher Zeit etwas tut.

Chandra und Toueg [35] haben eine eigene Terminologie für die Sicherheits- und Lebendigkeitseigenschaften von Fehlerdetektoren eingeführt. Sie nennen die oben angegebene Sicherheitseigenschaft *starke Genauigkeit* (*strong accuracy*) und die Lebendigkeitseigenschaft *starke Vollständigkeit* (*strong completeness*). Ein *perfekter Fehlerdetektor* (*perfect failure detector*) erfüllt starke Genauigkeit und starke Vollständigkeit, d.h. jeder lokale Fehlerdetektor macht keine falschen Verdächtigungen und erkennt nach endlicher Zeit jeden Absturz. Wenn man das asynchrone Modell erweitert, indem man lediglich annimmt, perfekte Fehlerdetektoren stünden zur Verfügung, ist *consensus* lösbar [35].

2.6.2 Unzuverlässige Fehlerdetektoren

Die Annahme, daß perfekte Fehlerdetektoren zur Verfügung stehen, ist in der Praxis oft nicht gerechtfertigt und schränkt das Fehlerdetektor-Systemmodell stark im Hinblick auf seine Realitätsnähe ein. Es stellt sich die Frage, ob es möglicherweise schwächere Annahmen über die Eigenschaften von Fehlerdetektoren gibt, unter denen *consensus* trotzdem lösbar ist.

Es gibt einen schematischen Weg, wie man die Eigenschaften perfekter Fehlerdetektoren abschwächen kann, indem man einfach andere Kombinationen der Quantoren ausprobiert. Aus der Quantifizierung "$\forall p : \forall q : \ldots$" im Rahmen der starken Genauigkeit könnte man alternativ auch schreiben "$\forall p : \exists q : \ldots$". Alle möglichen Kombinationen von Quantoren sind in Abbildung 2.4 dargestellt. Die Aussagen, welche quantifiziert werden (und die man im Geiste in Abbildung 2.4 immer ergänzen muß), lauten wie folgt:

- ...Es ist nie der Fall, daß p den Prozeß q verdächtigt ohne daß q tatsächlich abgestürzt ist (Sicherheit bzw. Genauigkeit).
- ...Falls q abstürzt, dann wird p den Prozeß q nach endlicher Zeit dauerhaft verdächtigen (Lebendigkeit bzw. Vollständigkeit).

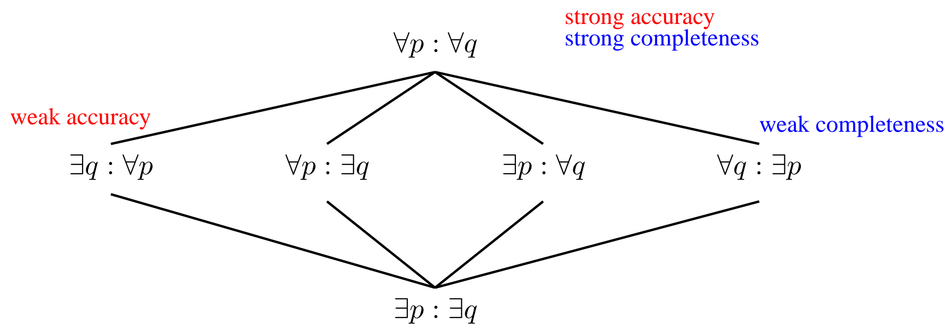


Abbildung 2.4: Varianten von Sicherheit (bzw. Genauigkeit) und Lebendigkeit (bzw. Vollständigkeit) bei Fehlerdetektoren.

Interessanterweise sind die Kombinationen $\exists q : \forall p$ bei der Sicherheit und $\forall q : \exists p$ bei der Lebendigkeit immernoch nützlich, um *consensus* zu lösen. Was bedeuten diese Abschwächungen?

- $\exists$ ein korrekter Prozeß q : $\forall p$: Es ist nie der Fall, daß p den Prozeß q verdächtigt ohne daß q tatsächlich abgestürzt ist (Sicherheit bzw. Genauigkeit).

  In Bezug auf die Sicherheit der Fehlerdetektoren bedeutet dies, daß es einen korrekten Prozeß q gibt, den niemand fälschlicherweise verdächtigt. Die Aussagen der lokalen Fehlerdetektoren in Bezug auf den Absturz aller anderen Prozesse (selbst die korrekten) müssen nicht stimmen. Diese Eigenschaft wird als *schwache Genauigkeit* (*weak accuracy*) bezeichnet [35]. Schwache Genauigkeit ist sogar nützlich, wenn sie (in Anlehnung an die zweite Form partieller Synchronität) erst nach endlicher Zeit gilt. Dies wird dann als *letztendlich schwache Genauigkeit* (*eventual weak accuracy*) bezeichnet.

- $\forall q : \exists$ ein korrekter Prozeß p : Falls q abstürzt, dann wird p den Prozeß q nach endlicher Zeit dauerhaft verdächtigen (Lebendigkeit bzw. Vollständigkeit).

  In Bezug auf die Lebendigkeit von Fehlerdetektoren bedeutet dies, daß es für jeden individuellen Prozeßabsturz einen anderen korrekten Prozeß p gibt, der letztendlich diesen Absturz auch entdeckt. Andere Prozesse (außer p) müssen den Absturz nicht entdecken. Diese Eigenschaft wird *schwache Vollständigkeit* (*weak completeness*) genannt [35]. Wenn es aber immer einen Prozeß gibt, der den Absturz erkennt, dann kann dieser Prozeß natürlich die Verdächtigung an alle anderen Prozesse weiterleiten. Wenn es also eine Möglichkeit gibt, zuverlässige Informationsverteilung (z.B. per *reliable broadcast*) zu betreiben, dann kann man einen schwach vollständigen Fehlerdetektor in einem stark vollständigen umwandeln.

Oben wurde bereits angedeutet, daß die Lebendigkeitseigenschaften unter *crash* jeweils auf die Menge der korrekten Prozesse eingeschränkt werden muß. Warum wird dies auch beim Existenzquantor der Sicherheitseigenschaft gemacht? Die Antwort auf diese Frage ist eng mit der Lösbarkeit von *consensus* verbunden. Ein Fehlerdetektor, der schwache Vollständigkeit und letztendlich schwache Genauigkeit erfüllt, wird *letztendlich schwacher Fehlerdetektor* (*eventually weak failure detector*) genannt. Chandra und Toueg [34, 35] stellten 1991 einen Algorithmus vor, der im asynchronen Systemmodell mit einem letztendlich schwachen Fehlerdetektor das *consensus*-Problem löst. Auf den ersten Blick mag dieses Resultat überraschen, weil es in diesem Modell sein kann, daß die Aussagen der lokalen Fehlerdetektoren einzelner Prozesse nie richtig sein müssen. Ihr Algorithmus ist allerdings sehr ähnlich zu dem, der bereits im Abschnitt 2.5.2 vorgestellt worden ist: Falsche Verdächtigungen gefährden nie die Sicherheit sondern nur die Lebendigkeit. Die letztendlich schwache Genauigkeit des Fehlerdetektors führt dazu, daß es nach endlicher Zeit einen Koordinator gibt, der nicht verdächtigt wird und eine Entscheidung durchsetzen kann. Hier wird die Notwendigkeit deutlich, die Sicherheitseigenschaft auf korrekte Prozesse zu beziehen (die Einschränkung ist interessanterweise symmetrisch: Bei der Lebendigkeit werden die aktiven Prozesse eingeschränkt, bei der Sicherheit die Menge der verdächtigten Prozesse). Die schwache Vollständigkeit (mit deren Hilfe starke Vollständigkeit simuliert wird) bewahrt die einzelnen Prozesse davor, möglicherweise unendlich lange auf die Nachricht eines abgestürzten Koordinators zu warten. Wie im Falle partieller Synchronität muß man in diesem Fall aber davon ausgehen, daß die Mehrheit der Prozesse das Protokoll überlebt.

Vergleichbarkeit von Fehlerdetektoren. Die Namensgebung des letztendlich schwachen Fehlerdetektors deutet bereits darauf hin, daß seine Eigenschaften schwächer sind als beispielsweise die Annahmen über einen perfekten Fehlerde-

tektor. Um aber auch die anderen Arten von Fehlerdetektoren miteinander vergleichen zu können, benötigt man eine genaue Definition von "schwächer als". Chandra und Toueg haben dafür das Konzept der *Reduzierbarkeit* (*reducibility*) vorgeschlagen [35]. Dazu werden alle Fehlerdetektoren mit denselben Eigenschaften zu Fehlerdetektorklassen zusammengefaßt. Eine solche Klasse F_1 ist *schwächer* als eine andere Klasse F_2 (geschrieben als $F_1 \leq F_2$), falls ein verteilter Algorithmus im asynchronen Systemmodell existiert, der mit Hilfe eines beliebigen Fehlerdetektors $f_2 \in F_2$ einen Fehlerdetektor $f_1 \in F_1$ simulieren kann.

Zum Beispiel ist schwache Vollständigkeit trivialerweise schwächer (im oben definierten Sinne) als starke Vollständigkeit: Ein stark vollständiger Fehlerdetektor kann einfach die Fehlerentdeckungen für alle außer einen Prozeß ignorieren; das Resultat ist ein schwach vollständiger Detektor. Auf der anderen Seite wurde oben bereits beschrieben, daß man einen schwach vollständigen Fehlerdetektor umwandeln kann in einen stark vollständigen, wenn beispielsweise ein *reliable broadcast* zur Verfügung steht. Da ein solcher Mechanismus in asynchronen Systemen implementiert werden kann [91], ist starke Vollständigkeit auch schwächer als schwache Vollständigkeit. In einem solchen Falle bezeichnet man beide Fehlerdetektorklassen als *äquivalent*. In asynchronen Systemen ist es also egal, ob man starke oder schwache Vollständigkeit annimmt. Ein Vergleich der Sicherheitseigenschaften (also der Genauigkeit) der Fehlerdetektoren im Hinblick auf die "schwächer als"-Relation ist in Abbildung 2.5 dargestellt. Mit Hilfe dieser Relation zeigten Chandra, Hadzilacos und Toueg [33], daß ein letztendlich schwacher Fehlerdetektor der schwächste Fehlerdetektor für *consensus* ist, d.h. jeder Fehlerdetektor, mit dem man *consensus* lösen kann, ist reduzierbar auf einen letztendlich schwachen Fehlerdetektor.

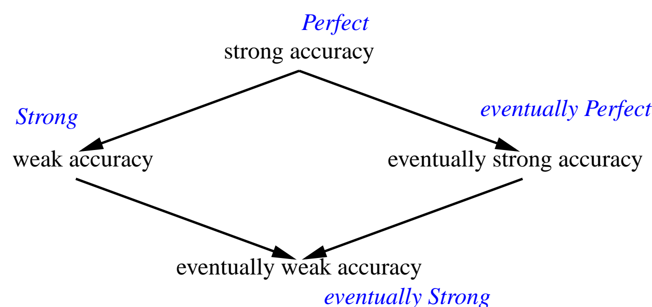


Abbildung 2.5: Vergleich der verschiedenen Genauigkeitsannahmen von Fehlerdetektoren mittels der "schwächer als"-Relation.

Weitere Forschungen auf dem Gebiet. War das Interesse an asynchronem *consensus* nach der Veröffentlichung des FLP-Theorems stark abgeflaut, führte das Konzept der unzuverlässigen Fehlerdetektoren zu einer neuen Welle von

Veröffentlichungen zu diesem Thema [3–5, 7–10, 23, 24, 33, 51, 52, 61–63, 65, 70, 71, 84–88, 92, 101, 103, 105, 124, 159, 166]. Die "klassischen" Fehlerdetektoren bezogen sich auf die *crash*-Fehlerannahme. Es gibt verschiedene Ansätze, das Fehlerdetektorkonzept zur Lösung von *consensus* unter anderen, schwereren Fehlerannahmen nutzbar zu machen. Beispiele sind *crash* in Verbindung mit verlustbehafteten Kanälen und/oder Partitionierungen [5,7,22,50,51] und *crash-recovery* [4,96,143].

Was ist ein letztendlich schwacher Fehlerdetektor? In synchronen Systemen mit festen Zeitschranken auf allen wesentlichen Systemparametern sind perfekte Fehlerdetektoren relativ einfach implementierbar. Umgekehrt kann man mit perfekten Fehlerdetektoren eine Art virtuelle Globalzeit einführen, indem man die Kommunikation in "synchrone" Runden gliedert. Die Rundennummer gibt den momentanen Wert der Uhr an. In jeder Runde schickt jeder Prozeß an jeden anderen Prozeß eine Nachricht. Auf einem Prozeß endet die Runde genau dann, wenn er von jedem anderen Prozeß entweder eine Nachricht erhalten hat, oder dieser Prozeß vom Fehlerdetektor als ausgefallen deklariert worden ist.

Letztendlich perfekte Fehlerdetektoren kann man in partiell synchronen Systemen implementieren [35], d.h. in Systemen, in denen Zeitschranken existieren ohne bekannt zu sein, oder wo Zeitschranken erst nach endlicher Zeit gelten (vgl. Abschnitt 2.5.2). Man kann letztendlich perfekte Fehlerdetektoren sogar implementieren, wenn sowohl das eine als auch das andere gilt (d.h. Zeitschranken existieren, sind unbekannt, *und* gelten erst nach endlicher Zeit). Wenn man demnach noch schwächere Fehlerdetektoren betrachtet, redet man also über Systeme, die weniger Annahmen über Zeitschranken machen als das partiell synchrone Systemmodell.

Fehlerdetektoren mit starker Vollständigkeit und schwacher Genauigkeit beispielsweise entdecken jeden Absturz, geben aber falsche Alarme über alle Prozesse mit Ausnahme eines einzigen korrekten Prozesses p. Daß sie jeden Absturz erkennen, ist einfach zu implementieren und benötigt keine expliziten Zeitschranken. In asynchronen Systemen ist es aber auf der anderen Seite unmöglich, einen Fehlalarm bezüglich eines bestimmten Prozesses zu vermeiden. Darum muß man für die schwache Genauigkeit beispielsweise annehmen, daß eine gültige (bekannte) Zeitschranke bezüglich der Kommunikation mit p existiert. Diese Zeitschranke ist aber asymmetrisch, d.h. sie muß nur für Nachrichten gelten, die p an alle anderen schickt und nicht für Nachrichten, die alle anderen an p schicken. Somit dürfte deutlich geworden sein, daß es relativ kompliziert ist, schwache Fehlerdetektoreigenschaften mit Echtzeitschranken in Verbindung zu setzen. Dies deutet darauf hin, daß das Fehlerdetektormodell bestimmte Zeitannahmen in feinerer Granularität und dem Anwendungsgebiet angemessener auszudrücken vermag [117].

2.7 Modell 3: Zeitbeschränktes asynchrones Modell

Neben der Einführung expliziter Zeitschranken und dem Fehlerdetektormodell gibt es noch weitere Systemmodelle, die bei der Validierung fehlertoleranter verteilter Anwendungen eingesetzt werden, von denen die beiden prominentesten in diesem und dem folgendem Abschnitt vorgestellt werden. Das erste dieser beiden Modelle wird *zeitbeschränktes asynchrones Systemmodell* (*timed asynchronous system model*) [45] genannt. Das zeitbeschränkte asynchrone Modell basiert wie das Fehlerdetektormodell auch auf den Grundannahmen des (reinen) asynchronen Modells. Allerdings wurde in das zeitbeschränkt asynchrone Modell ein expliziter Bezug zur Realzeit eingebaut. In diesem Modell haben Prozesse Zugriff auf eine eingebaute Hardware-Uhr. Es wird angenommen, daß der Gangunterschied zwischen dieser Uhr und einer gedachten genauen Realzeit durch eine bekannte Konstante beschränkt ist. Zwar bedeutet dies, daß man daraus eine Schranke auf den Gangunterschied zwischen zwei beliebigen Uhren im System berechnen kann, jedoch heißt das nicht, daß die Unterschiede der Ausführungsgeschwindigkeiten der einzelnen Rechner oder Nachrichtenlaufzeiten beschränkt sind. Im Gegenteil: Das Modell geht weiterhin davon aus, daß diese Werte unbeschränkt bleiben.

Die Fehlerannahme unterscheidet sich jedoch von der des Fehlerdetektormodells. Das zeitbeschränkt asynchrone Modell geht davon aus, daß die Kommunikation unzuverlässig ist, d.h. Nachrichten können verloren gehen. Wie im Fehlerdetektormodell können Prozesse abstürzen, aber im zeitbeschränkt asynchronen Modell können diese Prozesse auch wieder mit der Ausführung ihres lokalen Algorithmus beginnen (zur Sicherung von wichtigen Zustandsdaten steht stabiler Speicher zur Verfügung). Im Gegensatz zu allen anderen Modellen werden keine expliziten Schranken auf die Häufigkeit von Fehlern (sowohl von Abstürzen als auch von Nachrichtenverlusten) angenommen.

Durch die Zugriffsmöglichkeit auf eine Art Echtzeit ist es möglich, Echtzeitschranken auf der Ausführung bestimmter Netzwerkdienste zu definieren und deren Einhaltung zu überprüfen. Die Definition solcher Schranken ist für alle Dienste im zeitbeschränkt asynchronen Modell verpflichtend. Das bedeutet, daß man auf der einen Seite zwar nicht genau sagen kann, ob ein entfernter Prozeß abgestürzt ist oder nicht, auf der anderen Seite weiß man aber immerhin, wann eine Nachricht, die man von einem Prozeß erwartet, verspätet eintrifft. Dies wird *Fehlerbewußtsein* (*fail-awareness*) genannt [66].

Leider führt Fehlerbewußtsein alleine noch nicht dazu, daß man *consensus* im zeitbeschränkt asynchronen Modell lösen kann. Da es keine Schranken auf den Ausfallraten von Prozessen und Nachrichten gibt, ist es im allgemeinen sogar unmöglich, irgendwelche Lebendigkeitseigenschaften zu erreichen. Warum ist das zeitbeschränkt asynchrone Modell trotzdem ein gutes Modell für manche

Fehlertoleranzanwendungen?

Zunächst ist das zeitbeschränkt asynchrone Modell wegen der "Schwäche" seiner Annahmen sehr realistisch. Bei der Herleitung des Modells hat man sich auf eine aufwendige Reihe von Tests und Simulationen realer Netzwerke gestützt, deren Meßergebnisse im Detail im Originalartikel von Cristian und Fetzer [45] dokumentiert sind. Für die Realitätsnähe des Modells spricht auch die hohe Fehlererfassung, die durch den Verzicht auf Annahmen über Fehlerhäufigkeiten erreicht wird. Die Annahme, daß relativ genaue Hardware-Uhren zur Verfügung stehen, ist heutzutage ebenfalls keine echte Einschränkung mehr. Heute besitzt praktisch jeder Rechner eine Hardware-Quarz-Uhr, deren Gangunterschied zur Realzeit auf 10^{-4} Sekunden beziffert werden kann (d.h., pro Sekunde Realzeit geht die Uhr maximal eine Mikrosekunde zusätzlich vor oder nach).

Fortschrittsannahmen und bedingte Lebendigkeitseigenschaften. Cristian und Fetzer haben natürlich auch die Lösbarkeit von *consensus* im zeitbeschränkt asynchronen Modell untersucht [64]. Die Ergebnisse ihrer Untersuchung [45] wurden bereits in Abschnitt 2.5.2 erwähnt und beziehen sich auf ein lokales Netzwerk mit mehreren Workstations, die über ein handelsübliches Ethernet verbunden sind. Das untersuchte System alternierte zwischen langen Phasen der Stabilität und kurzen Phasen der Instabilität. Im Durchschnitt dauerte eine stabile Phase etwa 218 Sekunden und eine instabile etwa 340 Millisekunden. Stabile Phasen waren also im Durchschnitt 641 Mal länger als instabile Phasen. Allgemein ist es also realistisch, sogenannte *Fortschrittsannahmen* (*progress assumptions*) zu treffen. Fortschrittsannahmen formalisieren die eben angedeuteten Beobachtungen in der folgenden Form: "Es existiert eine Konstante c, so daß es unendlich oft eine stabile Phase des Systems gibt, die mindestens c Sekunden andauert." (Allgemeiner wird dies noch bezogen auf eine "stabile" Partition des Netzwerkes.) In anderen Worten heißt dies: Das System ist ausreichend oft synchron für mindestens c Sekunden.

Mit Hilfe solcher Fortschrittsannahmen ist es nun möglich, geforderte Lebendigkeitseigenschaften (wie die Terminierungseigenschaft von *consensus* beispielsweise) in folgender Art und Weite abzuwandeln: "Unter der Fortschrittsannahme S gilt die Lebendigkeit L." Eine Lebendigkeitseigenschaft L wird auf diese Art und Weise transformiert in eine sogenannte *bedingte Lebendigkeitseigenschaft* (*conditional timeliness property*) $S \Rightarrow L$. Die Terminierungsforderung von *consensus* wird dann zu:

- Wenn Fortschrittsannahme S gilt, dann wird jeder Prozeß, der nicht abstürzt, sich nach endlicher Zeit entscheiden.

Das Abschwächen der Lebendigkeit ist der Preis, den man für die hohe Fehlererfassung bezahlen muß, die im wesentlichen durch den Verzicht auf Schranken

auf Fehlerraten erreicht wird. Solche Schranken werden implizit durch die Fortschrittsannahme eingeführt. Im Fehlerdetektormodell waren solche Schranken (d.h., daß nicht mehr als t Prozesse abstürzen) explizit Teil des Modells. Dies heißt aber *nicht*, daß beide Formen der Beschränkung von Fehlerraten dasselbe ausdrücken: Ein Algorithmus, der mit diesen Schranken im Fehlerdetektormodell eine bestimmte Lebendigkeitseigenschaft erreicht, wird dies nicht immer im zeitbeschränkten asynchronen Modell schaffen. Der Grund liegt in dem explizit gegebenen Zeitintervall c. Der Algorithmus hat im Prinzip nur immer wieder c Sekunden Zeit, um einen gewissen Fortschritt in seiner Abarbeitung auf ein bestimmtes Ziel hin zu erreichen. Wenn er dies nicht innerhalb dieser Zeit schafft, wird er das Ziel der Lebendigkeit unter Umständen nie erreichen. Die Formulierung impliziter Schranken in Form von Fortschrittsannahmen hat demnach einen wichtigen Vorteil gegenüber expliziten Schranken wie im Fehlerdetektormodell: Im Fehlerdetektormodell wurde angenommen, daß das System "lange genug" stabil ist, damit der Algorithmus eine bestimmte Lebendigkeitseigenschaft erreichen konnnte; dies wurde mit "unendlicher Stabilität nach endlicher Zeit" approximiert. Fortschrittsannahmen hingegen geben genau an, wie lange das System stabil sein wird und in welchem zeitlichen Rahmen ein Algorithmus letztendlich Fortschritt erzielen muß. Fortschrittsannahmen schließen daher eine Lücke, die in der Literatur oft diskutiert worden [35, 49] und die für viele praktisch relevanten Szenarien von Bedeutung ist.

2.8 Modell 4: Das quasi-synchrone Modell

Als letztes Systemmodell für fehlertolerante Anwendungen wird das sogenannte *quasi-synchrone Systemmodell* (*quasi synchronous system model*) [11] vorgestellt. Wie der Name schon andeutet, liegt dieses Modell etwas näher am "synchronen Rand" des Modellspektrums als das zeitbeschränkte asynchrone Modell.

Grundannahmen und Lösbarkeit von *consensus*. Das quasi-synchrone Modell ist eigentlich ein voll synchrones Systemmodell. Es wird angenommen, daß feste und bekannte Schranken auf allen wesentlichen Netzwerkparametern existieren. Um das Modell etwas realistischer zu machen, sind diese Schranken jedoch sehr groß gewählt; die Wahrscheinlichkeit, daß diese Schranken von einem realen System eingehalten werden, wird als 1 angenommen. Die Fehlerannahme im quasi-synchronen Modell erlaubt Nachrichtenverluste und Prozeßabstürze, jedoch ebenfalls nur mit festen oberen Schranken. Es ist also keine Frage, daß *consensus* in diesem Systemmodell lösbar ist.

Zeitfehler und Zeitfehlerdetektoren. Aber wo liegt der Unterschied dieses Modells beispielsweise zum rein synchronen Modell? Das quasi-synchrone Modell erlaubt es, einfach kurzerhand kleinere Zeitschranken für manche oder sogar alle relevanten Netzwerkparameter anzunehmen. Natürlich steigt mit schwindenden Zeitschranken auch die Wahrscheinlichkeit, daß das Netzwerk diese Schranken nicht mehr einhält. Falls ein Ereignis in Bezug auf diese Zeitschranken zu spät stattfindet, spricht man von einem *Zeitfehler* (*timing failure*). Im schlimmsten Falle kann ein Zeitfehler (wie oben erwähnt) zu einer Verletzung der Sicherheit führen, z.B. wenn ein entfernter Prozeß zu Unrecht verdächtigt wird. Im quasi-synchronen Modell wird jedoch angenommen, daß man einen Zeitfehler immer zuverlässig entdecken kann. Der Mechanismus, mit dem dieses erreicht werden kann, wird *perfekter Zeitfehlerdetektor* (*perfect timing failure detector*) genannt. Ein derartiger Detektor ist wie folgt spezifiziert:

- Wenn ein Zeitfehler auftritt, dann wird er von allen überlebenden Prozessen innerhalb einer festen Echtzeitschranke erkannt.

- Es ist nie der Fall, daß ein Ereignis, welches innerhalb der angenommenen Zeitschranke stattfindet, als Zeitfehler erkannt wird.

Ein Zeitfehlerdetektor urteilt über stattfindende Ereignisse immer abhängig von den gewählten Zeitschranken. Eine Anwendung kann diese Zeitschranken zur Laufzeit dynamisch ändern. Dies ist günstig für die Effizienz vieler Anwendungen. Oft hängt die Leistungsfähigkeit mancher Anwendungen davon ab, wie schnell der Absturz eines entfernten Prozesses erkannt wird. Damit dies schneller geht, ist man daran interessiert, möglichst kurze Zeitschranken anzunehmen. Zu kurze Zeitschranken bergen jedoch das Risiko von Fehleinschätzungen. Dieser Fall kann durch die Informationen des Zeitfehlerdetektors ausgeschlossen werden. Eine Anwendung kann darum zugunsten der Effizienz zunehmend kürzere Zeitschranken annehmen und wenn nötig auf längere Zeitschranken umschalten, wenn beispielsweise die Anzahl der Zeitfehler ein bestimmtes Maß übersteigt [11]. Die Lebendigkeit von Algorithmen ist jedoch immer garantiert, da schlimmstenfalls die ungünstigsten, längsten Zeitschranken gewählt werden können und spätestens dann Fehlerdetektoren mit den Vollständigkeits- und Genauigkeitseigenschaften zur Verfügung stehen, die man zur Lösung von Fehlertoleranzproblemen benötigt.

Praktikabilität des quasi-synchronen Modells. Von den bisher vorgestellten Systemmodellen erscheint das quasi-synchrone Modell aus praktischer Sicht am problematischsten, da man zur Umsetzung von Systemen in diesem Modell einen perfekten Zeitfehlerdetektor implementieren muß. Offensichtlich ist der Bau eines solchen Detektors in Systemen wie dem Internet nicht möglich. Almeida, Veríssimo und Casimiro [11] argumentieren, daß es jedoch heute bereits viele

Netzwerktechnologien gibt, deren Echtzeitverhalten genau spezifiziert ist (wie beispielsweise ATM, ISDN oder GSM) und die für die Realisierung eines perfekten Zeitfehlerdetektors benutzt werden können. Es ist auch nicht notwendig, daß das gesamte Netzwerk echtzeitfähig ist. Um einen perfekten Zeitfehlerdetektor zu implementieren, genügt es, wenn nur ein kleiner Teil des Gesamtnetzwerkes diese Echtzeiteigenschaften besitzt (siehe Abbildung 2.6). Parallel zu den Nutzdaten auf einem "asynchronen" Basisnetzwerk kann man auf einem echtzeitfähigen Kontrollnetzwerk die Nutzdaten "vorankündigen". Ein Ausbleiben der Nutzdaten bis zu einem gewissen Zeitpunkt löst dann die Erkennung eines Zeitfehlers aus.

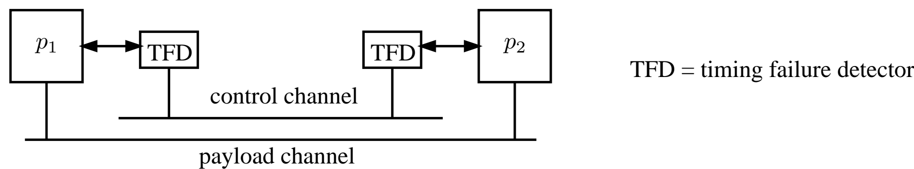


Abbildung 2.6: Zeitfehlererkennung unter Benutzung eines dezidierten Echtzeitkanals.

2.9 Zusammenfassung und Vergleich

Fehlertolerante Systeme kommen normalerweise dort zum Zuge, wo das Versagen eines Computersystems einen außerordentlichen Schaden anrichten würde. Demnach werden an die Funktionstüchtigkeit derartiger Systeme höhere Anforderungen als üblich gestellt. In vielen Bereichen wird verlangt, daß der zugrundeliegende Fehlertoleranzalgorithmus verifiziert worden ist, bevor das System zum Einsatz kommt. Voraussetzung für eine solche Untersuchung ist ein formales Systemmodell. Ein Systemmodell beschreibt in einer begrenzten Menge von Attributen und Regeln die Grundannahmen über die Ausführungsumgebung [162]. Die Kunst dabei ist, ein Modell zu finden,

1. welches alle relevanten Systemeigenschaften aus der Realität genau genug wiederspiegelt, und

2. mit dem sich die Kernprobleme eines bestimmten Anwendungsbereiches gut und einfach formulieren lassen.

Bei der Analyse fehlertoleranter reaktiver Systeme sind zunächst qualitative Aspekte ihres Verhaltens wichtiger als quantitative, d.h. Fragen wie "Terminiert der Algorithmus?" oder "Kann der Algorithmus in Zustand x kommen wenn ein Fehler auftritt?". Diese Eigenschaften sind durch die beiden Klassen von

Sicherheits- und Lebendigkeitseigenschaften gut beschreibbar. Bei der Formulierung dieser Eigenschaften muß aber immer eventuelles Fehlverhalten berücksichtigt werden. Auf der Ebene des Algorithmus kann man Fehler durch zusätzliche Programminstruktionen modellieren.

In diesem Kapitel wurden die vier wichtigsten Systemmodelle für fehlertolerante verteilte Systeme vorgestellt. Alle Modelle basieren auf dem sogenannten asynchronen (oder "zeitfreien") Modell. In diesem Modell gibt es keine Schranken auf Nachrichtenlaufzeiten oder den Unterschieden in den Ablaufgeschwindigkeiten einzelner Prozesse. Dieses Systemmodell ist jedoch erwiesenermaßen nicht gut geeignet für Fehlertoleranzsysteme, weil darin das in der Fehlertoleranz so wichtige Problem des *consensus* nicht gelöst werden kann. Die vier vorgestellten Modelle versuchen, diese Tatsache durch zusätzliche Annahmen in unterschiedlicher Weise zu umgehen:

- Modell 1 durch die Einführung von expliziten Zeitschranken.
- Modell 2 durch die Einführung von unzuverlässigen Fehlerdetektoren.
- Modell 3 durch die Annahme von Hardware-Uhren, zeitbeschränkten Diensten und bedingten Lebendigkeitsannahmen.
- Modell 4 durch die Einführung von expliziten, dynamisch adaptierbaren Zeitschranken und einem perfekten Zeitfehlerdetektor.

Vergleich der Systemmodelle. Das synchrone oder partiell synchrone Modell (Modell 1) ist relativ einfach zu verstehen, weil es das Zeitverhalten des Systems global festlegt. Es hat in den Anfängen der Fehlertoleranztheorie vielfältigen Einsatz in der Beschreibung und Analyse von Algorithmen gefunden [126].

Im Gegensatz zum synchronen Modell erlaubt das Fehlerdetektormodell (Modell 2) eine feinere Abstufung verschiedener Synchronitätsannahmen sowie die Trennung von Funktion und Mechanismus der Fehlererkennung. Das Konzept der Fehlerdetektoren ist zwar relativ schwierig zu verstehen und muß für jede Fehlerannahme neu angepaßt werden, doch ist es das einzige Modell, welches zumindest auf der "Benutzungsoberfläche" jeden Bezug zu einer Vorstellung von Echtzeit vermeidet. Das Fehlerdetektormodell ist deswegen zu einem der populärsten Systemmodelle für die Analyse und Verifikation von Fehlertoleranzalgorithmen geworden.

Einen präzisen Vergleich zwischen dem Fehlerdetektormodell (Modell 2) und dem zeitbeschränkt asynchronen Modell (Modell 3) hat Fetzer [63] durchgeführt. Um die unterschiedlichen Fehlerannahmen der Modelle anzugleichen, benutzt er als Grundlage für das Fehlerdetektormodell eine Weiterentwicklung von Aguilera,

Chen und Toueg [5]. In dieser Variante gilt für Prozesse das *crash-recovery*-Fehlermodell und Kanäle können Nachrichten verlieren. Fetzer zeigt, daß mit der Hinzunahme von Fortschrittsannahmen beide Modelle bis auf einen kleinen Unterschied in der Fehlerannahme auf Kanälen äquivalent sind. Ohne Fortschrittsannahmen fehlt dem zeitbeschränkten asynchronen Modell die Grundlage für die Lebendigkeit der in ihm ablaufenden Algorithmen. Durch die Hinzunahme von Hardware-Uhren und die notwendige Zeitbeschränkung aller Dienste ist es allerdings möglich, sogenanntes Fehlerbewußtsein herzustellen und wenigstens zu garantieren, daß das System in einem sicheren Zustand verbleibt. Das zeitbeschränkt asynchrone Systemmodell eignet sich darum vor allem für den Entwurf von Systemen, wo es eine "sichere Alternative zu normalem Arbeiten" gibt [155]. Dies ist beispielsweise der Fall in automatischen Steuerungssystemen für Züge. In diesem Anwendungsgebiet ist das zeitbeschränkt asynchrone Systemmodell auch bereits zum Einsatz gekommen [60]. Die Trennung von eigentlichem Systemmodell und Fortschrittsannahmen unterstreicht die praktische Ausrichtung dieses Modells.

Das quasi-synchrone Systemmodell (Modell 4) ist neben Modell 1 das "synchronste" Modell, welches hier vorgestellt worden ist. Die Fehlererfassung ist durch die Annahme von oberen Schranken auf den Fehlerhäufigkeiten nicht so hoch wie im zeitbeschränkt asynchronen Modell. Dafür ist es möglich, die Effizienz einer Datenübertragung relativ genau zu überwachen. Der zentrale Mechanismus hierfür ist der perfekte Zeitfehlerdetektor, der mit modernen Echtzeitnetzwerktechnologien realisiert werden kann. Das quasi-synchrone Modell ist das einzige, in dem man eine Art "Echtzeit-Lebendigkeit" erreichen und nachweisen kann. Es eignet sich dewegen insbesondere für fehlertolerante Anwendungen, die Echtzeitanforderungen erfüllen müssen [104, 178].

Zusammenfassung und Ausblick. Die in diesem Kapitel vorgestellten Systemmodelle sind bei der Betrachtung von Fehlertoleranzmethoden in verteilten Systemen unerläßlich. Sie bilden quasi die Diskussionsgrundlage, d.h. eine Basis, auf die man sich verständigen muß, bevor es mit der eigentlichen Arbeit losgehen kann. Die Darstellungsweise war dabei eher informal gestaltet. Im folgenden Kapitel wird die Grundlage der vorgestellten Systemmodelle noch etwas formaler präsentiert, um genaue Aussagen über die Wirkungsweise von Fehlertoleranzmethoden machen zu können. In Kapitel 4 wird dann auf das Fehlerdetektormodell zurückgegriffen, um eine Klasse von Algorithmen zum verteilten Beobachten zu analysieren und zu verifizieren.

Kapitel 3

Formale Grundlagen der Fehlertoleranz

3.1 Einführung

Die Literatur über fehlertolerante Computersysteme ist mannigfaltig: Mindestens ein halbes Dutzend wissenschaftliche Zeitschriften beschäftigen sich ausschließlich mit Fehlertoleranzmethoden und deren Validierungsmöglichkeiten, und es gibt etwa doppelt so viele jährlich stattfindende Konferenzen zu diesem Thema. Den entscheidenden Entwicklungsschub erlebte das Forschungsgebiet in den 1960er Jahren im Rahmen des US-amerikanischen Raumfahrtprogramms. In der Folge entwickelten sich hauptsächlich aus der Praxis heraus eine verwirrende Vielfalt von Terminologien und Konzepten, die Anfangs der 1990er Jahre selbst ausgewiesene Experten verunsicherte. Beispielsweise stellen Arora und Gouda in einem 1993 erschienenen Artikel fest [13]:

> [...] *the discipline* [*of fault-tolerance*] *itself seems to be fragmented.*

Wenigstens die Terminologiedebatte wurde etwas besänftigt, als 1992 das Buch "*Dependability: Basic Concepts and Terminology*" von Laprie [116] erschien. Es enthält einen in Gemeinschaftsarbeit mit vielen anderen Forschern festgelegten Terminologie- und Konzeptkatalog, der viele Bereiche der Fehlertoleranz abdeckt. Dennoch bleiben zentrale Konzepte der Fehlertoleranz, wie zum Beispiel das der *Redundanz*, unerwähnt.

Obwohl sich praktisch alle Forscher heute auf Laprie beziehen, bleibt der Einstieg in dieses Gebiet für Neulinge schwer. Es gibt mittlerweile durchaus lesenswerte Einführungen in Form von Büchern [55, 98, 119] oder Zeitschriftenartikeln [19, 44, 93], jedoch sind diese Texte noch weit entfernt von einer einheitlichen, konsistenten Darstellung des Gebietes. Es gibt viele Argumente dafür, daß

der Grund hierfür in den mangelnden formalen Ausrichtung des Gebietes liegt. Schließlich existiert noch keine gesicherte theoretische Basis vieler Fehlertoleranzverfahren.

Ziel und Beitrag dieses Kapitels. Ziel dieses Kapitels ist, die Ansätze einer solchen in der Literatur bereits existierenden Theorie aufzuzeigen und diese Theorie in einem wichtigen Punkt weiterzuentwickeln. Die Grundlagen stammen aus der 1998 veröffentlichten Fehlertoleranztheorie von Arora und Kulkarni [14, 15, 107]. Dieses Kapitel stellt die Theorie in den Grundzügen vor und entwickelt im Rahmen des darin verwendeten Systemmodells eine präzise Begrifflichkeit der Redundanz. Zwar gibt es bereits eine recht genaue Terminologie der Redundanz von Echtle [55], jedoch erscheint eine *formale* Definition von Redundanz und eine genauer Nachweis ihrer Rolle im Rahmen der Fehlertoleranz neu. Insbesondere trägt dieses Kapitel zur Präzisierung des oft ungenau verwendeten Begriffs der “Zeitredundanz” bei.

Ergebnisse dieses Kapitels im Überblick. Grundlage der Fehlertoleranztheorie von Arora und Kulkarni ist ein vollständig formales Systemmodell, in dem auch die Korrektheit eines Programms bezüglich einer Spezifikation genau definiert werden kann. Die Grundaussage der Theorie lautet:

> Ein fehlertolerantes Programm ist die Komposition aus einem fehlerintoleranten Programm und einer Menge von Fehlertoleranzkomponenten.

Es gibt nur zwei grundlegende Arten von solchen Fehlertoleranzkomponenten: Detektoren und Korrektoren (*detectors and correctors*). Detektoren sind Programmmodule, die feststellen, ob ein Prädikat auf dem Zustand des Gesamtsystems gilt oder nicht. Korrektoren sind Programmmodule, die einen bestimmten Systemzustand herstellen. Mittels dieser beiden Komponenten kann man tatsächlich eine große Menge an fehlertoleranten Systemen entwerfen. Bekannte Entwurfsmuster der Fehlertoleranz wie Schneiders *state machine approach* [161] sind gleichermaßen konstruierbar mittels Detektoren und Korrektoren.

Detektoren und Korrektoren sind nur über ihre Funktionalität definiert. Es bleibt also die Frage: Wie funktionieren diese Module grundsätzlich? Im Verlauf der Untersuchungen dieses Kapitels stellt sich heraus, daß nicht-erreichbare Zustände und Zustandsübergänge hierbei eine entscheidende Rolle spielen. Dies sind Zustände und Zustandsübergänge, die ein Programm im Normalfall (d.h. wenn keine Fehler auftreten) nicht benötigt. Die betrachteten Systemeigenschaften unterteilen sich in *Sicherheits-* und *Lebendigkeitseigenschaften* (im Sinne von *safety* und *liveness*, vgl. Kapitel 2). Es wird nachgewiesen:

1. Um Sicherheitsspezifikationen zu erfüllen, benötigt man nicht-erreichbare Zustände.
2. Um Lebendigkeitsspezifikationen zu erfüllen, benötigt man nicht-erreichbare Zustände und nicht-erreichbare Zustandsübergänge.

Nicht-erreichbare Zustände bilden die Grundlage für den neu definierten Begriff der *Speicherredundanz*. Im Rahmen der Theorie ist demnach Speicherredundanz notwendig, um Sicherheit zu erreichen. Nicht-erreichbare Zustandsübergänge bilden die Grundlage für den neu definierten Begriff der *Zeitredundanz*. Um Lebendigkeit zu erreichen, benötigt man demnach Zeitredundanz *und* Speicherredundanz.

Übertragen auf die Theorie von Arora und Kulkarni bedeutet dies:

1. Detektoren enthalten Speicherredundanz, und
2. Korrektoren enthalten Speicherredundanz und Zeitredundanz.

Diese Ergebnisse geben also Einblicke in die grundsätzliche Wirkungsweise von Fehlertoleranzkomponenten.

Ausblick. Zunächst werden in Abschnitt 3.2 die Grundzüge der Fehlertoleranztheorie von Arora und Kulkarni vorgestellt. In den folgenden drei Abschnitten wird ein vereinfachtes Systemmodell dieser Theorie entwickelt: Abschnitt 3.3 enthält die Beschreibung von fehlerfreien Systemen, während im Abschnitt 3.4 das Systemmodell um Fehlermöglichkeiten erweitert wird. Abschnitt 3.5 enthält eine formale Definition von Fehlertoleranz.

In den darauffolgenden Abschnitten wird die grundsätzliche Wirkungsweise von Fehlertoleranzmethoden untersucht. Dabei wird unterschieden, ob diese Methoden Sicherheit erreichen (Abschnitt 3.6) oder Lebendigkeit (Abschnitt 3.7). Diese Abschnitte enthalten die grundlegenden neuen Resultate. Abschnitt 3.9 diskutiert die Bedeutung dieser Resultate im Zusammenhang und Abschnitt 3.10 faßt die Ergebnisse des Kapitels nochmals kurz zusammen.

3.2 Die Fehlertoleranztheorie von Arora und Kulkarni

3.2.1 Detektoren und Korrektoren

Das Systemmodell der Fehlertoleranztheorie von Arora und Kulkarni basiert auf dem Konzept der *bewachten Anweisungen* (*guarded commands* [46]). Der Zu-

standsraum eines Programms ist dabei untergliedert in eine Menge von Variablen, und ein Programm besteht aus einer Menge von bewachten Anweisungen der Form:

$$\langle \text{guard} \rangle \rightarrow \langle \text{command} \rangle$$

Die *Wache* (*guard*) ist eine Boolesche Bedingung auf den Variablen des Programms. Die *Anweisung* (*command*) ist eine Zuweisung von Werten an die Variablen des Programms. Wenn die Wache in einem Zustand wahr ist, dann ist die bewachte Anweisung *aktiv*. Beginnend aus einem Startzustand wird ein Ablauf nach folgendem sich wiederholendem Schema konstruiert: Alle Anweisungen werden evaluiert, und aus der Menge der aktiven Anweisungen wird nichtdeterministisch eine Anweisung ausgewählt, deren Zuweisung ausgeführt wird. Dies ergibt den nächsten Zustand des Ablaufs, der auf die gleiche Art fortgesetzt wird. Falls mehrere Wachen wiederholt aktiv sind, sorgen sogenannte *Fairneßannahmen* [69] dafür, daß keine Anweisung unendlich oft vernachlässigt wird.

Fehler werden modelliert als eine Menge F von zusätzlichen bewachten Anweisungen. Die Ausführung eines Programms p unter Fehlereinflüssen ist dann definiert als die Ausführung der parallelen Komposition von p und F. Es wird angenommen, daß das Auftreten von Fehlern nach endlicher Zeit aufhört.

Die Theorie betrachtet ausschließlich Programme, die aus zwei Komponenten zusammengesetzt sind:

1. einem fehler-intoleranten Programm, d.h. einem Programm p, welches korrekt ist, wenn keine Fehler auftreten, und

2. einer Menge von Fehlertoleranzkomponenten.

Die Restriktion auf derartige Programme ist keine echte Einschränkung. Arora und Kulkarni zeigen, daß praktisch alle fehlertoleranten Programme auf diese Art und Weise dargestellt werden können. Sie tun dies, indem sie zwei grundlegende Arten von Fehlertoleranzkomponenten identifizieren:

- Ein *Detektor* ist eine abstrakte Programmkomponente, die feststellt, ob ein Prädikat X auf dem Zustand des Gesamtsystems gilt oder nicht. Dies wird angezeigt durch ein *Anzeigeprädikat* Z (*witness predicate*). Genauer ist ein *X-Detektor mittels Z* eine Programmkomponente, die unter anderem folgende Bedingungen erfüllt:

  1. Es wird nie Z angezeigt, wenn nicht X gilt (d.h. $Z \Rightarrow X$).
  2. Wenn X lange genug gilt, dann wird dies nach endlicher Zeit auch durch Z angezeigt.

- Ein *Korrektor* ist eine abstrakte Programmkomponente, die einen bestimmten Systemzustand herstellt und dies meldet. Genauer gesagt, ist ein *X-Korrektor mittels Z* eine Programmkomponente, die im wesentlichen folgende Eigenschaften erfüllt:

  1. Nach endlicher Zeit gilt X für immer (dargestellt in den temporallogischen Operationen aus Kapitel 2: $\Diamond\Box X$).
  2. Z zeigt nie fälschlicherweise an, daß X gilt (d.h. $Z \Rightarrow X$).
  3. Wenn X lange genug gilt, dann wird dies nach endlicher Zeit auch durch Z angezeigt.

3.2.2 Resultate

Die Theorie weist vier grundlegende Theoreme über Detektoren und Korrektoren nach. Die ersten beiden behandeln den Zusammenhang zwischen einem Programm p, das eine Sicherheitseigenschaft *SSPEC* (*safety specification*) erfüllen soll, auch wenn Fehler aus F auftreten:

1. Wenn F tolerierbar ist, dann kann man p mit Hilfe von Detektoren so verändern, daß p *SSPEC* erfüllt, auch wenn Fehler aus F auftreten. Mit anderen Worten: Detektoren sind hinreichend, um Sicherheitsspezifikationen einzuhalten. [107, S. 28]

2. Wenn p durch die Hinzufügung von Komponenten fehlertolerant bezüglich *SSPEC* gemacht wurde, enthält es Detektoren. Mit anderen Worten: Detektoren sind notwendig, um Sicherheitsspezifikationen einzuhalten. [107, S. 33]

Die weiteren Theoreme behandeln in ähnlicher Weise Fragen der Lebendigkeit. Angenommen, ein Programm p erfüllt eine Spezifikation *SPEC*. Wenn Fehler aus einer Fehlermenge F auftreten, wird dies in der Regel nicht mehr der Fall sein. Was muß man tun, damit p wenigstens nach endlicher Zeit wieder beginnt, gemäß *SPEC* zu funktionieren? Genauer gesagt: Was muß man tun, damit das System im Fehlerfall statt *SPEC* wenigstens $\Diamond$*SPEC* erfüllt? Die Antwort geben Theoreme 3 und 4:

3. Mittels Korrektoren kann man dafür sorgen, daß p nach endlicher Zeit wieder beginnt, gemäß *SPEC* zu funktionieren. Mit anderen Worten: Korrektoren sind hinreichend, um letztendlich wieder gemäß einer Spezifikation zu funktionieren.

4. Ein Programm, das ursprünglich eine Spezifikation verletzt und durch die Hinzufügung von Komponenten derart gestaltet wurde, daß es nach endlicher Zeit wieder gemäß der Spezifikation funktioniert, enthält Korrektoren. Mit anderen Worten: Korrektoren sind notwendig, um letztendlich wieder gemäß einer Spezifikation zu funktionieren.

3.2.3 Beispiel: *triple modular redundancy*

In diesem Abschnitt wird gezeigt, wie man bestehende Fehlertoleranzmechanismen mit Hilfe von Detektoren und Korrektoren modellieren kann. Betrachten wir ein System, welches aus drei redundanten Komponenten x, y und z besteht. Es wird gefordert, daß die Ausgabe einer korrekten Komponente auf einen Ausgang *out* geschaltet wird. (Solange noch kein Wert nach *out* zugewiesen wurde, hat *out* den Wert $\perp$.)

Genauer gesagt fordern wir die folgenden Sicherheits- und Lebendigkeitseigenschaften:

- Sicherheit: Es ist nie der Fall, daß *out* ein fehlerhafter Wert zugewiesen wird.
- Lebendigkeit: Nach endlicher Zeit wird ein Wert ungleich $\perp$ nach *out* zugewiesen.

Wenn keine Fehler auftreten, sind die Ausgaben der drei Komponenten identisch. Man kann darum eine beliebige Komponente auswählen (z.B. x) und deren Ausgang direkt nach *out* schalten. Die Basisfunktionalität kann darum erreicht werden durch das Programm:

$$out = \perp \quad \rightarrow \quad out := x$$

Wir betrachten nun eine Fehlermenge F, die dazu führt, daß der Ausgang von höchstens einer Komponente ein falsches Ergebnis ausgibt. Um die Sicherheitsspezifikation zu erfüllen, darf das Programm den Wert von x nicht durchschalten, wenn an x ein falsches Ergebnis anliegt. Dies ist erkennbar, wenn man das Prädikat $Z \equiv x = y \vee x = z$ überprüft. (Z ist demnach das Anzeigeprädikat für $X \equiv$ "x ist nicht fehlerhaft".) Man fügt dem fehlerintoleranten Programm also einen "X-Detektor mittels Z" zu, indem man die bewachte Anweisung nur ausführt, wenn die Konjunktion $X \wedge out = \perp$ zutrifft. Das folgende Programm erfüllt die geforderte Sicherheitseigenschaft, wenn Fehler aus F auftreten:

$$(x = y \vee x = z) \wedge out = \perp \quad \rightarrow \quad out := x$$

Leider erfüllt das Programm nun nicht mehr die Lebendigkeitsspezifikation, wenn Fehler aus F ausgerechnet die Komponente x in Mitleidenschaft ziehen. Wir benötigen also einen Korrektor, der den Zustand $X \equiv$ "*out* ist korrekt" herstellt. Der X-Korrektor besteht aus zwei Anweisungen, die jeweils eine korrekte Komponente nach *out* durchschalten:

$$\begin{array}{lcl} (y = z \vee y = x) \wedge out = \perp & \rightarrow & out := y \\ (z = x \vee z = y) \wedge out = \perp & \rightarrow & out := z \end{array}$$

Das vollständige Programm lautet demnach wie folgt:

$$\begin{array}{lcl} (x = y \vee x = z) \wedge out = \perp & \rightarrow & out := x \\ (y = z \vee y = x) \wedge out = \perp & \rightarrow & out := y \\ (z = x \vee z = y) \wedge out = \perp & \rightarrow & out := z \end{array}$$

Der dadurch realisierte Fehlertoleranzmechanismus ist in der Literatur unter dem Begriff *triple modular redundancy* (TMR) bekannt.

3.3 Programme, Spezifikationen und Korrektheit

Wir möchten gerne grundsätzliche Aussagen über die Funktionsweise von Detektoren und Korrektoren machen, insbesondere im Hinblick auf mögliche Definitionen von Redundanz. Dazu beginnen wir in diesem Abschnitt, die Basisdefinitionen des Systemmodells von Arora und Kulkarni vorzustellen.

Zustände und Abläufe. Gegeben sei eine nichtleere, abzählbare Menge C von *Zuständen*, die das System einnehmen kann. Ein *Zustandsprädikat über C* ist eine Teilmenge φ von C. Ein *Zustandsübergang über C* ist ein Paar (s, s') von Zuständen aus C, d.h. ein Element von $C \times C$. Ein Zustandsübergang wird auch als *Transition* bezeichnet. Ein *Ablauf über C* ist eine unendliche, nichtleere Folge $\sigma = s_1, s_2, s_3, \ldots$ von Zuständen aus C. Wir bezeichnen mit $\sigma|_i$ den Präfix der Länge i von σ, d.h. den endlichen Ablauf $s_1, s_2, \ldots, s_i$. Sei α ein Präfix und β ein Ablauf, dann ist $\alpha \cdot \beta$ die *Konkatenation* von α und β.

Eigenschaften: Sicherheit und Lebendigkeit. Eine *Eigenschaft über C* ist eine Menge P von Abläufen über C. Ein Ablauf σ *erfüllt die Eigenschaft P*, wenn $\sigma \in P$. Ablauf σ *verletzt die Eigenschaft P*, falls $\sigma \notin P$. Wir definieren nun die zwei wesentlichen Klassen von Eigenschaften: Sicherheits- und Lebendigkeitseigenschaften.

3.1 Definition (Sicherheitseigenschaft) Eine Sicherheitseigenschaft S über C ist eine Eigenschaft über C, die folgende Bedingung erfüllt: Für jeden Ablauf, der S verletzt, gibt es einen Präfix α, so daß für alle Fortsetzungen β der Ablauf $\alpha \cdot \beta$ die Eigenschaft S verletzt. Formal:

$$\sigma \notin S \Rightarrow \exists i . \forall \beta . \sigma|_i \cdot \beta \notin S$$

3.2 Definition (Lebendigkeitseigenschaft) Eine Lebendigkeitseigenschaft L über C ist eine Eigenschaft über C, die folgende Bedingung erfüllt: Für jeden endlichen Ablauf α existiert eine Fortsetzung β, so daß $\alpha \cdot \beta \in L$. Formal:

$$\forall i . \exists \beta . \sigma|_i \cdot \beta \in L$$

Verletzungen einer Sicherheitseigenschaft geschehen immer im Endlichen. Definition 3.1 charakterisiert also eine Menge von "unerwünschten" endlichen Abläufen. Im Gegensatz dazu besagt Definition 3.2, daß Lebendigkeitseigenschaften nur im Unendlichen verletzt werden können. Eine Lebendigkeitseigenschaft schließt also eine Menge von unendlichen Ablaufsuffixen aus. Alpern und Schneider [12] haben gezeigt, daß jede Menge von Abläufen (d.h. jede Eigenschaft) dargestellt werden kann als der Schnitt einer Sicherheits- und einer Lebendigkeitseigenschaft.

Einfache Programme. Programme werden normalerweise als Automaten formalisiert, d.h. ein Tupel (C, I, T) bestehend aus einer Zustandsmenge C, einer Menge von Startzuständen I und einer Menge von Zustandsübergängen T. Wir nennen ein solches Tupel ein *einfaches Programm*. Zur Darstellung solcher Programme verwenden wir Zustandsdiagramme. Als Beispiel zeigt Abbildung 3.1 ein Programm mit vier Zuständen a, b, c und d. Startzustände sind durch einen Stern gekennzeichnet und Zustandsübergänge durch einen Pfeil. Wir gehen davon aus, daß es keine *Endzustände* im eigentlichen Sinne gibt, d.h. Zustände, aus denen kein Pfeil hinausführt. Wenn ein Automat trotzdem einen solchen Endzustand s hat, nehmen wir einfach einen Zustandsübergang (s, s) in die Zustandsübergangsmenge auf. Wie im folgenden erläutert, macht diese Annahme die Betrachtung von Lebendigkeitseigenschaften eines Automaten leichter.

Lebendigkeitsannahmen und Programme. Ein einfaches Programm beschreibt eine Sicherheitseigenschaft, d.h. alle endlichen Abläufe über C, die ausgehend aus einem Zustand aus I nur Übergänge aus T benutzen. In einem Systemmodell, in dem man auch Aussagen über die Lebendigkeit eines Programms machen will, benötigt man zusätzlich noch eine Annahme über den möglichen Fortschritt des Automaten. Dies wird üblicherweise als *Fortschritts-* oder *Fairneßannahme* formalisiert. Wir wollen dieses Konzept als *Lebendigkeitsannahme* [181] bezeichnen.

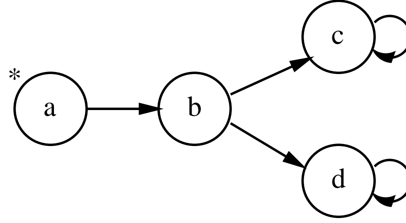


Abbildung 3.1: Beispiel für ein einfaches Programm.

3.3 Definition (Lebendigkeitsannahme) Sei $p = (C, I, T)$ ein einfaches Programm. Eine *Lebendigkeitsannahme* für p ist eine Lebendigkeitseigenschaft A über C, die folgende Bedingung erfüllt: Jeder endliche Ablauf α von p besitzt eine Fortsetzung β so daß gilt $\alpha \cdot \beta \in A$ und $\alpha \cdot \beta$ ist ein Ablauf von p.

Im Unterschied zu Lebendigkeitseigenschaften, die die bloße Existenz einer Ablauffortsetzung in der Menge fordern, muß eine Lebendigkeitsannahme die Fortsetzbarkeit durch p garantieren. Eine typische Lebendigkeitsannahme wird mit *Maximalität* bezeichnet und besagt: "Wann immer irgendein Zustandsübergang möglich ist, wird nach endlicher Zeit auch irgendein Zustandsübergang genommen." Durch diese Lebendigkeitsannahme schließt man alle endlichen Abläufe aus, die in einem Zustand enden, aus dem noch Zustandsübergänge herausführen.

Jede Lebendigkeitsannahme ist eine Lebendigkeitseigenschaft, aber nicht umgekehrt. Betrachten wir Abbildung 3.1 und die Lebendigkeitseigenschaft $L =$"nach endlicher Zeit d". Formal entspricht dies der Menge aller Abläufe, die irgendwann einmal den Zustand d enthalten. Beispielsweise sind die folgenden Abläufe in L:

$$d \cdot d \cdots, a \cdot d \cdot a \cdots, b \cdot d \cdot a \cdots, a \cdot b \cdot d \cdots, b \cdot a \cdot d \cdots$$

Diese Eigenschaft ist keine Lebendigkeitsannahme für das Programm aus Abbildung 3.1. Zwar kann jeder endliche Ablauf durch Hinzufügung von d zu einem Ablauf erweitert werden, der in L liegt. Jedoch ist dies nicht immer durch Zustandsübergänge des Programmes möglich (z.B. wenn sich das Programm bereits im Zustand c befindet). Eine Lebendigkeitsannahme A für das Programm wäre die oben bereits erwähnte Maximalität. Formal ist dies die Menge aller Abläufe, die entweder einen unendlichen Suffix von c's oder einen unendlichen Suffix aus d's haben. Die folgenden Abläufe sind beispielsweise Elemente von A:

$$d \cdot d \cdots, a \cdot d \cdot d \cdots, a \cdot b \cdot d \cdot d \cdots, c \cdot d \cdot c \cdot c \cdots, b \cdot a \cdot c \cdot c \cdots$$

3.4 Definition (Programm) Ein *Programm* ist ein Tupel $\Sigma = (C, I, T, A)$, wobei C eine Zustandsmenge, $I \subseteq C$ eine nichtleere Menge von Startzuständen, $T \subseteq C \times C$ eine nichtleere Menge von Zustandsübergängen über C und A eine Lebendigkeitsannahme für (C, I, T) ist.

Da man Lebendigkeitsannahmen schwer visualisieren kann, werden wir in den Abbildungen von Programmen die zum Programm gehörige Lebendigkeitsannahme (also z.B. Maximalität) als Text annotieren.

Die *Semantik eines Programms* Σ ist die Menge aller Abläufe, die es generieren kann. Wir bezeichnen diese Eigenschaft mit $prop(\Sigma)$. Formal ist dies der Schnitt aus der Lebendigkeitsannahme A und der Sicherheitseigenschaft, die durch (C, I, T) definiert ist. Statt $\sigma \in prop(\Sigma)$ schreiben wir manchmal verkürzend auch $\sigma \in \Sigma$.

Bezug zu anderen Formalismen. Wie oben bereits angedeutet, verwenden Arora und Kulkarni die Notation der bewachten Anweisungen für die Darstellung von Programmen. Im Vergleich zu der Notation der Zustandsdiagramme liegen diese Programme auf einer höheren Abstraktionsebene. Der Zustandsraum ist durch Variablen vorstrukturiert, bewachte Anweisungen entsprechen Mengen von Zustandsübergängen.

variables: $x \in \{1, 2, 3\}$ **initially** 1
actions:
$x = 1 \rightarrow x := 2$
$x = 2 \rightarrow x := 1$
$x = 2 \rightarrow x := 2$
$x = 2 \rightarrow x := 3$
$x = 3 \rightarrow x := 3$

Abbildung 3.2: Ein Programm in der Darstellung der bewachten Anweisungen.

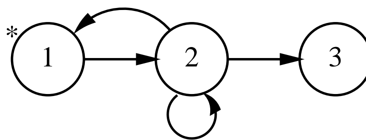


Abbildung 3.3: Das Programm aus Abbildung 3.2 als Zustandsdiagramm.

Abbildung 3.2 zeigt ein Beispielprogramm in der Notation der bewachten Anweisungen, dessen Zustandsdiagramm in Abbildung 3.3 dargestellt ist. Da es nur eine einzige Variable gibt, beschreiben die bewachten Anweisungen jeweils eine einzelne Transition. Die implizite Definition eines Ablaufs als wiederkehrende Evaluierung der Wachen und Auswahl einer aktiven Anweisung definiert die Lebendigkeitsannahme der Maximalität. Im Zustand $x = 2$ gibt es aber eine *Konflikt* zwischen drei Anweisungen. Maximalität alleine würde nun Abläufe erlauben, die einen unendlichen Suffix mit Zustand $x = 2$ haben, bzw. unendlich zwischen Zustand $x = 1$ und $x = 2$ hin- und herspringen.

Starke und schwache Fairneß. Nichtdeterminismus bei der Wahl der Anweisung wird üblicherweise dazu benutzt, um nebenläufige Programmausführung zu modellieren. Die in Konflikt stehenden Anweisungen sind dann Instruktionen paralleler Prozesse. Aus diesem Blickwinkel macht es Sinn, *faire Konfliktauflösung* zu postulieren. Es gibt in der Literatur zwei prominente Arten von Fairneß [69]:

1. *Schwache Fairneß* bedeutet, daß eine Anweisung, die "lange genug" entlang eines Ablaufes aktiv ist, letztendlich auch ausgewählt wird.

2. *Starke Fairneß* bedeutet, daß eine Anweisung, die "oft genug" entlang eines Ablaufs aktiv ist, letztendlich auch ausgewählt wird.

Starke Fairneß ist eine Obermenge von schwacher Fairneß (d.h. jeder stark faire Ablauf ist auch schwach fair). Schwache Fairneß sondert Abläufe aus, in denen eine Anweisung nicht ausgeführt wird, obwohl sie unendlich oft *hintereinander* aktiv war. In unserem Beispiel ist der Ablauf $x = 1, 2, 2, 2, \ldots$ nicht schwach fair. Bei starker Fairneß muß die Anweisung nicht unendlich lange hintereinander aktiv sein, sondern es reicht aus, wenn sie unendlich oft aktiv ist (sie kann auch zwischendurch einmal nicht aktiv sein). In unserem Beispiel ist der Ablauf $x = 1, 2, 1, 2, 1, 2, \ldots$ nicht stark fair. Starke und schwache Fairneß sind Lebendigkeitsannahmen im Sinne von Definition 3.3. Eine Lebendigkeitsannahme ist umso stärker, je mehr "unfaire" Abläufe sie ausschließt.

In der *guarded commands*-Notation beziehen sich Lebendigkeitsannahmen auf bewachte Anweisungen. Man kann aber auch für die Notation der Zustandsdiagramme stärkere Lebendigkeitsannahmen als Maximalität definieren, wenn man jede Transition als eigene bewachte Anweisung interpretiert.

Temporallogische Formeln. Aufbauend auf dem intuitiven Verständnis aus Abschnitt 2.2.3 definieren wir nun temporallogische Formeln und die Eigenschaften, die sie repräsentieren. Wir beschränken uns in dieser Arbeit auf die Operatoren $\Box$ ("immer") und $\Diamond$ ("nach endlicher Zeit"). Die atomaren Bausteine dieser Formeln sind Zustandsprädikate sowie deren Booleschen Verknüpfungen.

3.5 Definition (Temporalformeln und deren Semantik) Sei Z eine Menge von Zustandsprädikaten. Temporalformeln sind wie folgt definiert:

1. Jedes Element $z \in Z$ ist eine Temporalformel.

2. Ist Φ eine Temporalformel, dann sind auch $\Diamond\Phi$ und $\Box\Phi$ Temporalformeln.

Sei φ ein Zustandsprädikat, Φ eine Temporalformel und $\sigma = s_1, s_2, s_3, \ldots$ ein Ablauf. Wir definieren:

1. σ erfüllt φ, falls $s_1 \in \varphi$.
2. σ erfüllt $\Diamond\Phi$, falls ein i existiert, so daß $\sigma' = s_i, s_{i+1}, \ldots$ erfüllt Φ.
3. σ erfüllt $\neg\Phi$, falls σ nicht erfüllt Φ.
4. Die weiteren Booleschen Operatoren sind analog definiert.
5. σ erfüllt $\Box\Phi$, falls σ erfüllt $\neg\Diamond\neg\Phi$.

Die Semantik einer Temporalformel Φ ist die Menge aller Abläufe, die Φ erfüllen.

Spezifikationen und Korrektheit. Eine Spezifikation beschreibt gewünschtes Systemverhalten und ist ebenfalls eine Eigenschaft, d.h. eine Menge von Abläufen. In Anlehnung an Arora und Kulkarni [14] beschränken wir uns auf sogenannte *fusions-abgeschlossene Eigenschaften* (*fusion closed properties*).

3.6 Definition (Fusions-Abgeschlossenheit) Sei C ein Zustandsraum, $s \in C$, X eine Eigenschaft, α und γ endliche Abläufe und β und δ unendliche Abläufe.

Eine Eigenschaft X ist genau dann *fusions-abgeschlossen*, wenn gilt: Falls $\alpha \cdot s \cdot \beta \in X$ und $\gamma \cdot s \cdot \delta \in X$ dann sind auch $\alpha \cdot s \cdot \delta$ und $\gamma \cdot s \cdot \beta$ in X.

Definition 3.6 besagt folgendes: Wenn ein Ablauf in einem bestimmten Zustand s angelangt ist, dann fällt die Entscheidung über das weitere Fortschreiten allein aufgrund von Informationen, die in s vorliegen. Wenn also zwei Abläufe der gegebenen Form in der Spezifikation liegen, dann muß auch das "*crossover*" in der Spezifikation sein (siehe Abbildung 3.4). Ein Beispiel für eine Eigenschaft, die nicht fusions-abgeschlossen ist, lautet: "Zustand x tritt nur auf falls vorher irgendwann Zustand y aufgetreten ist."

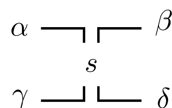


Abbildung 3.4: Schaubild zur Fusions-Abgeschlossenheit.

3.7 Definition (Spezifikation) Eine *Spezifikation über C* ist eine fusions-abgeschlossene Eigenschaft *SPEC* über C.

Die Beschränkung auf fusions-abgeschlossene Eigenschaften ist keine wirkliche Einschränkung. Alle herkömmlichen Programmiersprachen (wie C oder Java) sind fusions-abgeschlossen. Man kann zudem jede nicht-fusionsabgeschlossene Spezifikation durch Hinzufügung von zusätzlichen Variablen (*history variables*) fusions-abgeschlossen machen. In Anlehnung an Alpern und Schneider [12] ist es

darum möglich, jede Spezifikation als den Schnitt einer Lebendigkeitseigenschaft und einer fusions-abgeschlossenen Sicherheitseigenschaft darzustellen. Wir nennen diese beiden Teile *Sicherheits-* und *Lebendigkeitsspezifikation*.

Ein Programm Σ *erfüllt* eine Spezifikation $SPEC$, falls alle Abläufe von Σ in $SPEC$ enthalten sind. Andernfalls sagen wir, daß Σ die Spezifikation $SPEC$ *verletzt*.

Man kann die Eigenschaft eines Systems Σ, also $prop(\Sigma)$, auch als Spezifikation betrachtet, denn $prop(\Sigma)$ ist immer fusionsabgeschlossen.

3.8 Proposition Für alle Programme Σ ist $prop(\Sigma)$ fusionsabgeschlossen.

Beweis

BEWEISIDEE: Die Menge $SPEC = prop(\Sigma)$ besteht aus allen Abläufen, die Σ generieren könnte. Betrachtet man zwei Abläufe $\alpha \cdot s \cdot \beta$ und $\gamma \cdot s \cdot \delta$, dann sind die Übergänge nach und weg von s normale Zustandsübergänge von Σ. Demnach ist sowohl $\alpha \cdot s \cdot \delta$ durch Σ konstruierbar, als auch $\gamma \cdot s \cdot \beta$. Hierbei ist wichtig, daß die Lebendigkeitsannahme von Σ die Gestaltung dieser Abläufe nicht einschränkt.

Die Definition des "Bewahrens" einer Spezifikation (*maintains*) ist nützlich, um über Sicherheitsspezifikationen zu sprechen.

3.9 Definition (Bewahrung einer Spezifikation) Sei α ein endlicher Ablauf über C und $SPEC$ eine Spezifikation. Wir sagen, α *bewahrt* $SPEC$ gdw. es existiert eine Fortsetzung β so daß $\alpha \cdot \beta \in SPEC$.

Man kann eine Sicherheitsspezifikation mittels Definition 3.9 charakterisieren.

3.10 Proposition Die folgenden beiden Aussagen sind äquivalent:

1. $SPEC$ ist eine Sicherheitsspezifikation.
2. $\sigma \notin SPEC \Leftrightarrow$ es existiert ein Präfix α von σ so daß gilt: α bewahrt nicht $SPEC$.

Beweis

BEWEISIDEE: Der Beweis folgt sofort aus Definition 3.1.

Eine andere leicht zu beweisende Eigenschaft der Bewahrungsrelation ist folgende:

3.11 Proposition Für einen endlichen Ablauf σ sind die beiden folgenden Aussagen äquivalent:

1. σ bewahrt *SPEC*.

2. Alle Präfixe von σ bewahren *SPEC*.

Beweis

BEWEISIDEE: Der Beweis folgt aus mehrfacher Anwendung von Definition 3.9. Alle Schritte in diesem Beweis sind Äquivalenzen.

1 $\langle 1 \rangle 1.$ σ bewahrt *SPEC*.
BEWEIS: Folgt aus Teil 1. □
2 $\langle 1 \rangle 2.$ $\exists \delta . \sigma \cdot \delta \in SPEC$
BEWEIS: Anwendung von Definition 3.9. □
3 $\langle 1 \rangle 3.$ $\exists \delta . \alpha \cdot \beta \cdot \delta \in SPEC$
BEWEIS: Schreibe σ als $\alpha \cdot \beta$. □
4 $\langle 1 \rangle 4.$ $\exists \gamma . \alpha \cdot \gamma \in SPEC$
BEWEIS: Schreibe $\beta \cdot \delta$ als γ. □
5 $\langle 1 \rangle 5.$ α bewahrt *SPEC*.
BEWEIS: Anwendung von Definition 3.9. □
6 $\langle 1 \rangle 6.$ Q.E.D.
BEWEIS: Da keine Einschränkungen bezüglich α gemacht wurden, ergibt sich Teil 2 der Proposition. □

Bisher wurden die Begriffe immer unter der Annahme benutzt, daß keine Fehler im System auftreten. In den folgenden Abschitten geht es um Fehlermöglichkeiten und darum, was es bedeutet, korrekt unter Fehlereinflüssen zu sein.

3.4 Fehlermodellierung

Wir greifen die in Abschnitt 2.4 besprochene Idee auf und modellieren Fehler als unerwünschtes, zusätzliches Programmverhalten.

3.12 Definition (Fehlermodell) Sei $\mathcal{T}$ die Menge aller Programme. Ein *Fehlermodell* ist eine Abbildung $F : \mathcal{T} \rightarrow \mathcal{T}$. Falls $F((C, I, T, A)) = (C', I', T', A')$ müssen die folgenden Bedingungen gelten:

1. $C' = C$

2. $I' = I$

3 $T' \supseteq T$

4 $A' \supseteq A$

Ein Fehlermodell wird manchmal auch *Fehlerannahme* genannt. Gilt $F(\Sigma) = \Sigma'$ wird Σ' die *fehlerbehaftete Version* von Σ genannt. Manchmal bezeichnen wir mit $F(T)$ und $F(A)$ die Transitionsmenge und Lebendigkeitsannahmen der fehlerbehafteten Version.

In unserer Definition fügen Fehlermodelle keine neuen Zustände hinzu und erweitern auch nicht die Menge der initialen Zustände. Wir schließen also Fehler aus, die "vor dem Einschalten" auftreten. Fehler können aber einerseits neue Zustandsübergänge hinzufügen (Bedingung 3) oder die Lebendigkeitsannahme abschwächen (Bedingung 4). Im letzteren Fall muß A' natürlich weiterhin eine Lebendigkeitsannahme für das veränderte Programm sein. Abbildung 3.5 zeigt ein einfaches Programm und dessen fehlerbehaftete Version. Das Fehlermodell hat zwei zusätzliche Zustandsübergänge eingefügt, die als gestrichelte Pfeile dargestellt sind.

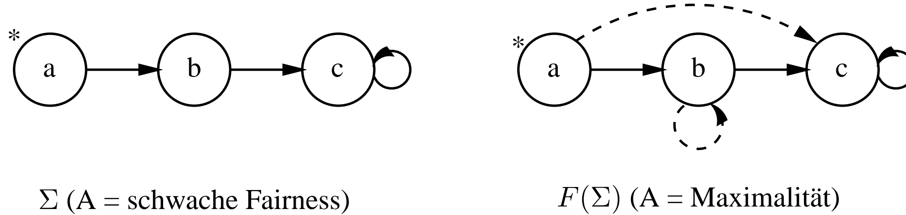


Abbildung 3.5: Beispiel für ein Programm und dessen fehlerbehaftete Version. Die Lebendigkeitsannahme von $F(\Sigma)$ wurde abgeschwächt auf Maximalität.

Implizit enthalten in Definition 3.12 ist die Tatsache, daß ein Fehlermodell F nichts am ursprünglichen Verhalten des Programmes ändert, d.h. alle Abläufe von Σ sind auch Abläufe von $F(\Sigma)$. Diese Eigenschaft wird in der Literatur als *Verhaltenserweiterung* bezeichnet (*enlargening behavior property*) [75]. Verhaltenserweiterung drückt aus, daß Fehler auftreten *können* und nicht auftreten *müssen*. Wenn Fehler auftreten müssen, besteht kein Unterschied mehr zwischen Fehleraktionen und Programmaktionen. Jede sinnvolle Definition eines Fehlermodells sollte demnach verhaltenserweiternd sein. Es ist leicht zu zeigen, daß Definition 3.12 diese Eigenschaft erfüllt.

3.13 Proposition (Fehlermodelle sind verhaltenserweiternd) Sei Σ ein Programm und F ein Fehlermodell. Es gilt:

$$prop(F(\Sigma)) \supseteq prop(\Sigma)$$

Beweis

ANNAHME: Σ ist ein Programm und F ein Fehlermodell.

ZEIGE: Jeder Ablauf von Σ ist ein Ablauf von $F(\Sigma)$.

BEWEIS: Folgt direkt aus der Tatsache, daß $T \subseteq F(T)$ und $A \subseteq F(A)$. □

Die Fehlermodellierung durch Definition 3.12 ist in einem gewissen Sinne vollständig: Abgesehen von fehlerhaften Anfangszuständen kann man jedes gewünschte Fehlverhalten durch eine Funktion F "implementieren". Fehlverhalten wird definiert als die Verletzung einer Spezifikation.

3.14 Definition (verletzbare Spezifikation) Sei $\Sigma = (C, I, T, A)$ ein Programm. Eine *verletzbare Spezifikation* von Σ ist eine Spezifikation $SPEC$, die folgenden Bedingungen genügt:

$$SPEC \subset I \cdot C^*$$

Hierbei ist $I \cdot C^*$ die Menge aller Abläufe, die mit einem Zustand aus I beginnen und dann beliebig fortgesetzt werden dürfen.

Definition 3.14 beschreibt präzise die Menge aller Spezifikationen, die man durch Fehlermodelle gemäß Definition 3.12 verletzen kann. Die Definition schließt natürlich aus, daß $SPEC = I \cdot C^*$ gewählt wird, da jedes System, welches aus I startet, diese Eigenschaft erfüllt.

3.15 Theorem (Vollständigkeit der Fehlermodellierung) Sei $SPEC$ eine verletzbare Spezifikation und Σ ein Programm, welches $SPEC$ erfüllt. Dann existiert ein F, so daß $F(\Sigma)$ die Spezifikation $SPEC$ verletzt.

Beweis

ANNAHME: 1. $SPEC$ ist eine verletzbare Spezifikation,

2. Σ ist ein Programm, welches $SPEC$ erfüllt.

ZEIGE: Es existiert ein F, so daß $F(\Sigma)$ $SPEC$ verletzt.

BEWEISIDEE: Die Beweisidee basiert auf der folgenden Beobachtung: Verletzungen der Sicherheit kann man ausschließlich durch die Hinzunahme von Zustandsübergängen erreichen, während Verletzungen der Lebendigkeit durch eine Kombination von zusätzlichen Zustandsübergängen und zusätzlichen "unfairen" Abläufen erreicht werden können.

1 $\langle 1\rangle 1.$ Es existiert ein Ablauf $\sigma \in I \cdot C^*$, der in $SPEC$ liegt, aber kein Ablauf von Σ ist.

BEWEIS: Existenz folgt aus Voraussetzung, daß $SPEC$ eine verletzbare Spezifikation ist. □

2 $\langle 1\rangle 2.$ Es existiert ein endlicher Präfix $\alpha \cdot s$ von σ, den man in $SPEC$ und Σ fortsetzen kann, d.h. für den eine Fortsetzung β existiert, so daß gilt: $\alpha \cdot s \cdot \beta \in SPEC$ und $\alpha \cdot s \cdot \beta \in \Sigma$.

BEWEIS: Aus der Definition einer verletzbaren Spezifikation folgt, daß σ mindestens einen Initialzustand besitzt, der auch ein Startzustand von Σ ist. □

3 $\langle 1 \rangle 3$. Bezeichne mit $t_1 = (s, s_1)$ die Transition, die auf s in σ folgt und mit $t_2 = (s, s_2)$ die Transition, die in $\alpha \cdot s \cdot \beta$ auf s folgt.
ANNAHME: $t_2 \notin T$
ZEIGE: Q.E.D.

BEWEIS: Definiere $F(T) := T \cup \{t_2\}$. □

4 $\langle 1 \rangle 4$. ANNAHME: $t_2 \in T$
ZEIGE: Q.E.D.

BEWEIS: Definiere $F(A)$ als die stärkste Lebendigkeitsannahme, die $A \cup \{\sigma\}$ enthält. □

5 $\langle 1 \rangle 5$. Q.E.D.

BEWEIS: Schritte $\langle 1 \rangle 3$ und $\langle 1 \rangle 4$ decken alle Fälle ab. Man kann dadurch in $F(\Sigma)$ den kompletten Ablauf σ nachspielen. Daraus folgt, daß $F(\Sigma)$ *SPEC* verletzt. □

Der Beweis von Theorem 3.15 ist konstruktiv in dem Sinne, daß man zu einem Programm und einer verletzbaren Spezifikation immer ein Fehlermodell angeben kann, welches die Spezifikation verletzt. Als Beispiel betrachten wir den Automaten in Abbildung 3.6 (links), der im fehlerfreien Fall die Spezifikationen $SSPEC = \Box\neg f$ und $LSPEC = \Box\Diamond d$ erfüllt. Die Lebendigkeitsannahme von Σ sei starke Fairneß.

Um die Sicherheitsspezifikation *SSPEC* zu verletzen, genügt es, einen Zustandsübergang einzuführen, der direkt oder indirekt den Weg zum Zustand f ebnet. In der fehlerbehafteten Version von Σ ist dies die Transition von b nach e. Um die Lebendigkeitsspezifikation *LSPEC* zu verletzen, genügt es, die Lebendigkeitsannahme auf Maximalität abzuschwächen. Jetzt ist ein unendliches Verweilen in Zustand c möglich, und damit die Verletzung von *LSPEC* gegeben.

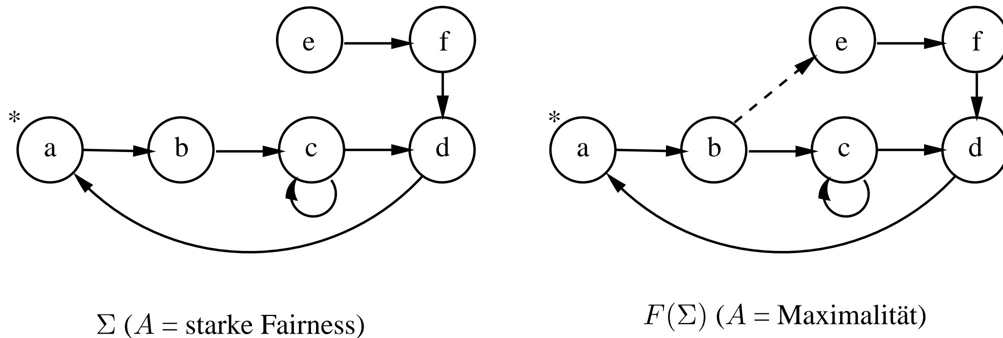


Abbildung 3.6: Beispiel zur Konstruktion von F. Die Lebendigkeitsannahme von Σ ist starke Fairneß, die von $F(\Sigma)$ ist zu Maximalität abgeschwächt worden.

3.5 Fehlertoleranz

3.5.1 Definitionen von Fehlertoleranz

Informal bedeutet Fehlertoleranz, daß ein System ein definiertes Verhalten aufweist, wenn Fehler auftreten. In der Literatur hat es eine Reihe von verschiedenen formalen Definitionen gegeben, die wir nun kurz zusammenfassen.

Eine der bekanntesten Definitionen stammt von Arora und Gouda [13]. Dort wird angenommen, daß die Fehlerannahme durch eine Transformation F gegeben ist. Ein System Σ ist fehlertolerant für eine Spezifikation SPEC, wenn nicht nur Σ sondern auch $F(\Sigma)$ SPEC erfüllt (das Programm besitzt also die Fähigkeit, die Fehler aus F zu tolerieren). Später haben Arora und Kulkarni [14] diese Definition noch verfeinert. Ein System ist fehlertolerant, wenn $F(\Sigma)$ eine "schwächere" Spezifikation $\mathit{SPEC}' \supseteq \mathit{SPEC}$ erfüllt [121]. Die Spezifikation SPEC' beschreibt demnach, welches Fehlverhalten noch "akzeptabel" ist und welches nicht. Die Spezifikation SPEC' wird *fehlerbezogene Spezifikation* genannt (*tolerance specification* [14,73], *failure mode* [142], *fault assumption* [58], *failure semantics* [44,122] oder *fault-affected behavior* [123]).

In der Praxis gibt es eine Reihe von fehlerbezogenen Spezifikationen, die oft wiederkehren. Häufig wird zum Beispiel $\mathit{SPEC}' = \mathit{SPEC}$ gewählt. Das bedeutet, das System erfüllt auch im Fehlerfall seine ursprüngliche Spezifikation. Dies wird in der Literatur als *maskierende Fehlertoleranz* (*masking fault-tolerance* [14,142,158]) bezeichnet. Wenn die fehlerbezogene Spezifikation schwächer ist als die ursprüngliche Spezifikation, spricht man von *fail-softness* [142] oder *graceful degradation* [94].

Wie eine entsprechend abgeschwächte Spezifikation aussehen sollte, ist im allgemeinen applikationsabhängig [94]. Entsprechend der Unterteilung in Sicherheits- und Lebendigkeitsspezifikation kann man aber zwei applikationsunabhängige Arten von Fehlertoleranz unterscheiden. Erfüllt ein System im Fehlerfall weiterhin seine Sicherheits-, verletzt aber seine Lebendigkeitsspezifikation, spricht man von *fail-safe-Fehlertoleranz* [14]. In diesem Falle ist also SPEC' gleich der Sicherheitsspezifikation. Diese Art von Fehlertoleranz ist immer dann nützlich, wenn es eine "sichere Alternative zum normalen Arbeiten" gibt [155]. Beispiele sind hier Steuerungssysteme im Zugverkehr. Tritt eine schwere Störung auf, kann man die Sicherheit des verbleibenden Zugverkehrs erreichen, indem man alle Signale auf rot stellt.

Man könnte eine dritte Form von Fehlertoleranz definieren, bei der ein System im Fehlerfall zwar die Sicherheitsspezifikation verletzt, hingegen aber die Lebendigkeitsspezifikation einhält. In der Praxis gibt es jedoch keine Beispiele, die zeigen, daß die bloße Einhaltung der Lebendigkeit eine sinnvolle Korrektheits-

annahme im Fehlerfall ist. Durchaus sinnvoll hingegen ist folgendes: Wie üblich bezeichnen wir die Sicherheits- und Lebendigkeitsspezifikation im fehlerfreien Fall mit *SSPEC* und *LSPEC*. Man definiert die fehlerbezogene Spezifikation nun als $SPEC' = \Diamond SSPEC \cap LSPEC = \Diamond SPEC$. Dies wird als *nicht-maskierende Fehlertoleranz* (*non-masking fault tolerance* [14]) bezeichnet. Statt also "immer" sicher zu sein, ist man nun "nach endlicher Zeit" sicher. Das System erfüllt also keine Sicherheitseigenschaft mehr, sondern nur noch eine "modifizierte" Lebendigkeitseigenschaft. Sinnvoll ist diese Art von Fehlertoleranz in Situationen, in denen maskierende Fehlertoleranz nicht (oder nur zu einem hohen Preis) erreicht werden kann. In unbemannten Weltraumsonden kann beispielsweise kosmische Strahlung dazu führen, daß der flüchtige Speicher des Systems einen annähernd zufälligen Wert annimmt. Die Sicherheitsspezifikation kann also "nicht immer" garantiert werden. Man kann die Systeme jedoch so konstruieren, daß sie nach endlicher Zeit wieder einen sicheren Zustand erreichen und wieder wie gewünscht funktionieren. Tabelle 3.1 faßt die besprochenen Arten von Fehlertoleranz zusammen.

		Sicherheitsspezifikation erfüllt?	
		Ja	Nein
Lebendigkeitsspezifikation	Ja	maskierend	nicht-maskierend
erfüllt?	Nein	*fail-safe*	keine

Tabelle 3.1: Verschiedene Arten von Fehlertoleranz.

Kompositionelle Sichtweise. Arora und Goudas Definition (die mit den Definitionen von Liu und Joseph [122, 123] sowie Weber [182] übereinstimmt) bezieht sich immer auf ein komplettes System auf einer bestimmten Abstraktionsstufe. Nordahl [142] hingegen unterscheidet auf einer gegebenen Abstraktionsebene zwischen Komponenten und ihrem Zusammenwirken. Bei Nordahl besteht ein System aus einer Menge von Subsystemen und einem sogenannten *design*, einer Beschreibung, wie diese Komponenten zusammenwirken. Die Subsysteme sind gegeben durch eine Menge von Komponentenspezifikationen. Fehler können dazu führen, daß Komponenten nicht mehr ihre Komponentenspezifikation einhalten. Ein Fehlermodell F gibt dann an, welche Kombinationen von abgeschwächten Komponentenspezifikationen erlaubt sind.

Ein *design* D ist fehlertolerant für eine Spezifikation *SPEC*, wenn die Komposition der Komponentenspezifikationen mittels D *SPEC* erfüllt, und zwar für jede durch F erlaubte Kombination. (Diese Definition wird auch implizit von Schepers [158] benutzt.)

Ein Beispiel für dieses Verständnis ist ein *hot-standby*-System bestehend aus zwei replizierten Servern A_1 und A_2, die beide gemeinsam einen Dienst erbringen,

der durch eine Spezifikation S beschrieben ist. Normalerweise läuft Server A_1, jedoch besagt die Fehlerannahme F, daß maximal ein Server abstürzen kann. Wenn A_2 abstürzt, bleibt alles beim alten. Um jedoch den Absturz von A_1 zu tolerieren, benötigt man eine zusätzliche Komponente, die die Koordination zwischen A_1 und A_2 in diesem Fall übernimmt. Wenn diese Komponente den Absturz von A_1 entdeckt, wird Server A_2 angewiesen, die Kommunikation mit der Außenwelt zu übernehmen. Eine Beschreibung der Wirkungsweise dieser Koordinationskomponente ist das *design*. Um zu validieren, daß das *design* fehlertolerant ist, muß man zeigen, daß das zusammengesetzte System in allen Fällen S erfüllt, d.h. wenn beide Server korrekt funktionieren oder wenn nur noch einer von beiden korrekt funktioniert.

Rushby [155] hat in einem lesenswerten Überblicksartikel diese beiden Klassen von Fehlertoleranzdefinitionen beschrieben. Die Definitionen des ersten Typs (also Arora und Gouda [13] und andere) nennt er *berechnend* (*calculational approaches*), da die Auswirkungen von Fehlern auf die Systemspezifikation berechnet werden müssen. Die kompositionellen Ansätze (also Nordahl [142] und andere) nennt Rushby *spezifizierend* (*specification approaches*). In ihnen geht man davon aus, daß die Auswirkungen von Fehlern auf Komponentenspezifikationen bereits gegeben sind. Wie an anderer Stelle gezeigt [75], sind beide Ansätze komplementär. In dieser Arbeit werden wir den berechnenden Ansatz verwenden.

3.5.2 Fehlertolerante Versionen

Es gibt verschiedene Möglichkeiten, fehlertolerante Systeme zu entwerfen. Ausgangspunkt ist in der Regel eine Spezifikation *SPEC* des gewünschten Verhaltens und eine Annahme F über die zu tolerierenden Fehler. Eine häufig gewählte Vorgehensweise besteht aus zwei Schritten:

1. Man nimmt eine fehlerfreie Umgebung an und entwirft mit bekannten Standardmethoden ein Programm p, welches *SPEC* erfüllt.

2. Man fügt Fehlertoleranzkomponenten zu p hinzu. Die Wahl dieser Komponenten hängt von F ab.

Diese Vorgehensweise hat viele Vorteile. Erstens eröffnet sich die Möglichkeit der Wiederverwendung von existierenden Programmen im ersten Schritt. Zweitens ist es möglich, die Fehlertoleranzmethoden mehr oder weniger automatisch (durch etablierte Ingenieursmethodik oder maschinelle Transformationen) und damit verläßlich in ein System einzubringen. Schließlich zeigt drittens die in Abschnitt 3.2 beschriebene Theorie, daß alle praktisch relevanten Fehlertoleranzmechanismen durch die Komposition eines fehler-intoleranten Systems mit Fehlertoleranzkomponenten beschrieben werden können.

Um diese Vorgehensweise zu formalisieren, benötigen wir ein Konzept, welches ausdrückt, daß ein System Σ_1 im fehlerfreien Fall dasselbe tut wie ein anderes System Σ_2.

3.16 Definition (Erweiterung eines Programmes) Programm $\Sigma_2 = (C_2, I_2, T_2, A_2)$ *erweitert* Programm $\Sigma_1 = (C_1, I_1, T_1, A_1)$ (geschrieben $\Sigma_2 \geq \Sigma_1$) falls folgende Bedingungen gelten:

1. $C_2 = C_1$
2. $A_2 = A_1$
3. $prop(\Sigma_2) = prop(\Sigma_1)$

Der in Punkt 3 verwendete Gleichkeitsbegriff wird in der Literatur manchmal als *Spurengleichheit* bezeichnet. Für die spätere Verwendung von Definition 3.16 ist es wichtig, daß sich Punkt 3 nur auf das fehlerfreie Verhalten von Σ_1 und Σ_2 bezieht, d.h. nur auf Zustände und Transitionen, die von initialen Zuständen aus erreichbar sind. Mittels Definition 3.16 kann man jetzt das Entwurfsziel von Fehlertoleranzprojekten definieren.

3.17 Definition (fehlertolerante Version) Seien Σ_1 und Σ_2 Programme, F ein Fehlermodell und $SPEC$ eine Spezifikation. Programm Σ_2 ist die F-tolerante Version von Σ_1 für $SPEC$ falls

1. Σ_1 erfüllt $SPEC$,
2. $F(\Sigma_1)$ verletzt $SPEC$,
3. Σ_2 erweitert Σ_1, und
4. $F(\Sigma_2)$ erfüllt $SPEC$.

Ein fehler-intolerantes Programm Σ_1 verletzt die eigene Korrektheitsbedingung $SPEC$, wenn Fehler aus F auftreten. Eine fehlertolerante Version Σ_2 macht im fehlerfreien Fall genau dasselbe wie Σ_1, erfüllt aber auch dann $SPEC$, wenn Fehler aus F auftreten.

Abbildung 3.7 zeigt ein Beispiel für Definition 3.17. Sei $SSPEC = \Box\neg f$ und $LSPEC = \Box\Diamond d$. Das System Σ_1 (in Abbildung 3.7 links, ohne den gestrichelten Pfeil) erfüllt offensichtlich sowohl $SSPEC$ als auch $LSPEC$. Das Fehlermodell F fügt eine Transition von b nach e ein. Dies führt dazu, daß $F(\Sigma_1)$ beide Korrektheitsbedingungen verletzt. Nun kann man aus Σ_1 ein Programm Σ_2 konstruieren (in Abbildung 3.7 rechts, ohne den gestrichelten Pfeil), welches im Normalfall dasselbe Verhalten ausweist wie Σ_2, aber im Gegensatz zu Σ_1 auch

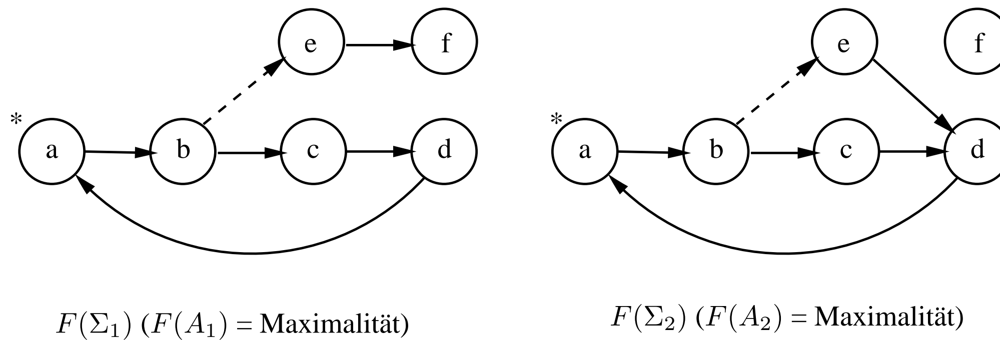


Abbildung 3.7: Beispiel für eine fehlertolerante Version.

noch bezüglich *SSPEC* und *LSPEC* korrekt ist, wenn Fehler auftreten. Das System Σ_2 ist also die F-tolerante Version von Σ_1 für *SSPEC* und *LSPEC*. Zu beachten ist, daß *SSPEC* fusionsabgeschlossen ist.

3.6 Fehlertoleranzmechanismen für Sicherheit

Wir betrachten nun Sicherheitsspezifikationen und untersuchen Methoden, mit denen man Sicherheitsspezifikationen auch im Fehlerfall einhalten kann.

Zunächst geht es in Abschnitt 3.6.1 um Konzepte, mit denen man erreichen kann, daß ein Fehlermodell zwei ähnliche Programme "gleich behandelt". Anschließend untersuchen wir in Abschnitt 3.6.2, welche Art von Fehlern zu einer Verletzung einer Sicherheitsspezifikation führen können und welche dieser Fehler nicht tolerierbar sind. Abschnitt 3.6.3 ist der Kern dieses Abschnitts: Dort wird *Speicherredundanz* definiert und nachgewiesen, daß diese Form von Redundanz notwendig ist, um Sicherheitsspezifikationen einzuhalten.

3.6.1 Sicherheitskonservative Fehlermodelle

Die Definition eines Fehlermodells basiert auf Funktionen, die Programme transformieren. Ein Fehlermodell kann zwei "ähnliche" Programme Σ_1 und Σ_2 ganz unterschiedlich behandeln und ganz unterschiedliches Fehlverhalten in beiden möglich machen. Wenn man jedoch über fehlertolerante Versionen spricht, geht man in der Regel davon aus, daß F in Σ_2 "dasselbe" Fehlverhalten einbaut wie in Σ_1. Man benötigt also ein Konzept, die "Ähnlichkeit" von Fehlverhalten formal beschreiben zu können.

Eine mögliche Definition wäre die folgende: Wenn $\Sigma_2 \geq \Sigma_1$, dann $F(\Sigma_2) \geq F(\Sigma_1)$. Peled und Joseph [146, p.103] nennen diese Eigenschaft *Monotonie* (sie verwenden jedoch einen anderen Erweiterungsbegriff). Wie die folgende Proposition zeigt,

ist dies jedoch eine sehr starke Annahme, denn sie steht im Konflikt mit Definition 3.17.

3.18 Proposition Seien Σ_1 und Σ_2 Programme, F ein Fehlermodell, *SPEC* eine Spezifikation und sei Σ_2 die F-tolerante Version von Σ_1 für *SPEC*. Dann gilt $F(\Sigma_2) \not\supseteq F(\Sigma_1)$.

Beweis

ANNAHME: Σ_2 ist die F-tolerante Version von Σ_1 für *SPEC*.

ZEIGE: $F(\Sigma_2) \not\supseteq F(\Sigma_1)$

BEWEISIDEE: Der Beweis ist indirekt: Wir führen die Annahme "$F(\Sigma_2)$ erweitert $F(\Sigma_1)$" zum Widerspruch, indem wir zeigen, daß in diesem Fall $F(\Sigma_2)$ *SPEC* verletzen würden.

1 ⟨1⟩1. ANNAHME: $F(\Sigma_2)$ erweitert $F(\Sigma_1)$.
ZEIGE: falsch

1.1 ⟨2⟩1. Jeder Ablauf von $F(\Sigma_2)$ ist auch ein Ablauf von $F(\Sigma_1)$ und umgekehrt.
BEWEIS: Folgt aus der Annahme aus Schritt ⟨1⟩1 und der Definition 3.16. □

1.2 ⟨2⟩2. Es gibt einen Ablauf $\sigma_1 \in F(\Sigma_1)$, der nicht in *SPEC* liegt.
BEWEIS: Folgt aus der Voraussetzung, daß $F(\Sigma_1)$ *SPEC* verletzt (dies ist Teil von Definition 3.17). □

1.3 ⟨2⟩3. Es gibt einen Ablauf $\sigma_2 \in F(\Sigma_2)$, für den gilt $\sigma_2 = \sigma_1$.
BEWEIS: Folgt aus Schritt ⟨2⟩2 und Schritt ⟨2⟩1. □

1.4 ⟨2⟩4. Es gibt einen Ablauf σ_2 von $F(\Sigma_2)$, der nicht in *SPEC* liegt.
BEWEIS: Folgt aus Schritt ⟨2⟩3 und Schritt ⟨2⟩2. □

1.5 ⟨2⟩5. Q.E.D.
BEWEIS: Schritt ⟨2⟩4 impliziert, daß $F(\Sigma_2)$ *SPEC* verletzt. Dies ist ein Widerspruch zu der Voraussetzung, daß Σ_2 eine F-tolerante Version von Σ_1 für *SPEC* ist. □

2 ⟨1⟩2. Q.E.D.
BEWEIS: Folgt aus Schritt ⟨1⟩1. □

Nach Proposition 3.18 dürfen wir also nicht annehmen, daß Σ_2 das Programm Σ_1 auch unter F erweitert. Andernfalls gäbe es keine fehlertolerante Version von Σ_1. Wir benötigen also eine schwächere Form von Monotonie.

Zum Ziel bringt uns die folgende Überlegung: Wir möchten formalisieren, daß F in Σ_2 dasselbe Fehlverhalten einbaut wie in Σ_1. Wenn Σ_2 eine Erweiterung von Σ_1 ist, haben beide also eine gemeinsame Zustandsmenge C. Wenn in Σ_1 also eine zusätzliche Transition $(s, s') \in C \times C$ eingebaut wird, muß diese Transition auch in Σ_2 eingebaut werden.

3.19 Definition (sicherheitskonservatives Fehlermodell) Seien $\Sigma_1 = (C_1, I_1, T_1, A_1)$

und $\Sigma_2 = (C_2, I_2, T_2, A_2)$ Programme mit $\Sigma_2 \geq \Sigma_1$. Ein Fehlermodell F heißt *sicherheitskonservativ* falls gilt:

$$t \in F(T_1) \setminus T_1 \Leftrightarrow t \in F(T_2) \setminus T_2$$

3.6.2 Tolerierbare und nicht tolerierbare Fehler

Zunächst geht es darum, die Klasse von Fehlern zu charakterisieren, die zur Verletzung einer Sicherheitsspezifikation führen können. Hier spielt die Fusionsabgeschlossenheit eine wichtige Rolle. In Anlehnung an ein Resultat von Arora und Kulkarni [15, Lemma 3.2] zeigen wir, daß man unter den gegebenen Voraussetzungen eine Menge von "schlechten" Transitionen identifizieren kann, d.h. Transitionen, die — wann immer sie auftreten — zu einer Verletzung der Sicherheitsspezifikation führen.

3.20 Lemma Sei $\Sigma = (C, I, T, A)$ ein Programm und *SSPEC* eine Sicherheitsspezifikation. Falls Σ *SPEC* verletzt und alle $x \in I$ *SPEC* bewahren, dann existiert ein Zustandsübergang $t \in T$ für den gilt: Für alle Abläufe σ von Σ, falls t in σ auftritt, ist $\sigma \notin$ *SSPEC*.

Beweis

ANNAHME:
1. *SSPEC* ist eine Sicherheitsspezifikation (d.h. eine fusionsabgeschlossene Sicherheitseigenschaft),
2. $\Sigma = (C, I, T, A)$ ist ein Programm, welches *SSPEC* verletzt, und
3. $\forall x \in I$ gilt, daß x *SSPEC* bewahrt.

ZEIGE: Es existiert eine Transition $t \in T$ so daß für alle Abläufe $\sigma \in \Sigma$ gilt: Wenn t in σ vorkommt, dann $\sigma \notin$ *SSPEC*.

BEWEISIDEE: Da *SSPEC* eine Sicherheitsspezifikation ist, gibt es in jedem Ablauf, der *SSPEC* verletzt, einen Punkt, an dem die Verletzung offenbar wird. Es bleibt zu zeigen, daß die dort auftretende Transition eine "schlechte" Transition ist, d.h. immer sofort zu einer Verletzung von *SSPEC* führt, wenn sie in einem beliebigen Ablauf auftritt. Dies wird unter Rückgriff auf die Fusionsabgeschlossenheit von *SSPEC* bewiesen.

1 ⟨1⟩1. Es existiert ein Ablauf σ von Σ der nicht in *SSPEC* liegt.
BEWEIS: Folgt aus Voraussetzung 2. □

2 ⟨1⟩2. σ kann geschrieben werden als $\alpha \cdot d \cdot b \cdot \beta$ so daß $\alpha \cdot d$ *SSPEC* bewahrt und alle Fortsetzungen von $\alpha \cdot d \cdot b$ *SSPEC* nicht bewahren.
BEWEIS: Voraussetzung 3 garantiert, daß es einen nichtleeren Präfix von σ gibt, der *SSPEC* bewahrt (schlimmstenfalls ist dies der Präfix, der nur aus dem initialen Zustand besteht). Der Rest dieses Schrittes folgt aus Schritt ⟨1⟩1 und der Voraussetzung, daß *SSPEC* eine Sicherheitseigenschaft ist. □

3 ⟨1⟩3. Betrachte einen anderen Ablauf ρ von Σ, in dem die Transition (d, b) auftritt, also $\rho = \hat{\alpha} \cdot d \cdot b \cdot \hat{\beta}$. Dann gilt $\rho \notin SSPEC$.

3.1 ⟨2⟩1. ANNAHME: $\rho \in SSPEC$
ZEIGE: falsch

3.1.1 ⟨3⟩1. Alle Präfixe von ρ bewahren *SSPEC*.
BEWEIS: Folgt aus der Annahme von Schritt ⟨2⟩1, der Voraussetzung, daß *SSPEC* eine Sicherheitseigenschaft ist, und Propositionen 3.10 und 3.11. □

3.1.2 ⟨3⟩2. $\hat{\alpha} \cdot d \cdot b$ bewahrt *SSPEC*.
BEWEIS: Folgt aus Schritt ⟨3⟩1 und der Tatsache, daß $\hat{\alpha} \cdot d \cdot b$ ein Präfix von ρ ist. □

3.1.3 ⟨3⟩3. $\exists \hat{\delta} : \hat{\alpha} \cdot d \cdot b \cdot \hat{\delta} \in SSPEC$.
BEWEIS: Folgt aus Schritt ⟨3⟩2 und Definition 3.9. □

3.1.4 ⟨3⟩4. $\exists \delta : \alpha \cdot d \cdot \delta \in SSPEC$.
BEWEIS: Folgt aus Schritt ⟨1⟩2 ($\alpha \cdot d$ bewahrt *SSPEC*) und Definition 3.9. □

3.1.5 ⟨3⟩5. $\alpha \cdot d \cdot b \cdot \hat{\delta} \in SSPEC$.
BEWEIS: Folgt aus Schritten ⟨3⟩3 und ⟨3⟩4 und der Fusionsabgeschlossenheit von *SSPEC*. □

3.1.6 ⟨3⟩6. $\alpha \cdot d \cdot b$ bewahrt *SSPEC*.
BEWEIS: Folgt aus Schritt ⟨3⟩5 und Definition 3.9. □

3.1.7 ⟨3⟩7. Q.E.D.
BEWEIS: Schritt ⟨3⟩6 ergibt einen Widerspruch zu Schritt ⟨1⟩2 (daß nämlich $\alpha \cdot d \cdot b$ *SSPEC nicht* bewahrt). □

3.2 ⟨2⟩2. Q.E.D.
BEWEIS: Folgt indirekt aus Schritt ⟨2⟩1. □

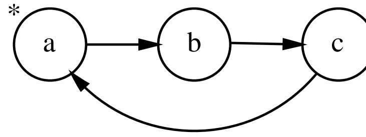
4 ⟨1⟩4. Q.E.D.
BEWEIS: Folgt direkt aus Schritt ⟨1⟩3. □

Oft werden fusionsabgeschlossene Sicherheitsspezifikationen durch eine Menge von "schlechten" Zuständen charakterisiert, die nie eingenommen werden dürfen. Lemma 3.20 macht deutlich, daß dies eine ungenaue Charakterisierung ist. Als Beispiel genügt ein Programm Σ mit drei Zuständen wie in Abbildung 3.8. Die Sicherheitseigenschaft, die durch Σ definiert wird, beschreibt alle endlichen Abläufe der folgenden Form:

$$a \cdot b \cdot c \cdot a \cdots$$

Diese Eigenschaft ist gemäß Proposition 3.8 fusionsabgeschlossen. Man kann sie also als Sicherheitsspezifikation *SSPEC* von Σ verwenden. Um *SSPEC* zu verletzen kann man nicht einfach in einen "schlechten" Zustand springen, weil es keine solchen Zustände von Σ gibt. Mann kann dies aber durch eine zusätzliche "schlechte" Transition (etwa von a nach c) erreichen. Zusätzliche Transitionen sind also *hinreichend*, um Sicherheitsspezifikationen zu verletzen. Das folgende

Lemma zeigt, daß zusätzliche Transitionen hierzu auch *notwendig* sind. Man kann sich also bei Sicherheitsbetrachtungen auf Fehlermodelle beschränken, die Transitionen hinzufügen.



A = Maximalität

Abbildung 3.8: Beispiel für ein System ohne "schlechte" Zustände.

3.21 Definition (Hinzufügung einer Transition) Sei F ein Fehlermodell, $\Sigma = (C, I, T, A)$ ein Programm und bezeichne $F(\Sigma) = \Sigma' = (C, I, T', A')$. Fehlermodell F *fügt eine Transition zu* Σ *hinzu* falls $T' \supset T$.

3.22 Lemma Sei F ein Fehlermodell, $\Sigma = (C, I, T, A)$ ein Programm, *SSPEC* eine Sicherheitsspezifikation und bezeichne $F(\Sigma) = \Sigma' = (C, I, T', A')$. Falls Σ *SSPEC* erfüllt und $F(\Sigma)$ *SSPEC* verletzt, dann fügt F mindestens eine Transition zu Σ hinzu.

Beweis

ANNAHME:
1. F ist ein Fehlermodell, $\Sigma = (C, I, T, A)$ und $\Sigma' = (C', I', T', A')$ sind Programme mit $F(\Sigma) = \Sigma'$.
2. *SSPEC* ist eine Sicherheitsspezifikation.
3. Σ erfüllt *SSPEC*.
4. Σ' verletzt *SSPEC*.

ZEIGE: Es existiert eine Transition $t' \in T'$, die durch F zu Σ hinzugefügt wurde.

BEWEISIDEE: Der Beweis ist indirekt: Wir nehmen an, F fügt keine Transition hinzu und zeigen, daß jeder Ablauf von Σ' ein Ablauf von Σ ist. Der Widerspruch ergibt sich dann aus der Tatsache, daß Σ' *SSPEC* verletzt.

1 $\langle 1 \rangle 1.$ ANNAHME: F fügt keine Transition zu Σ hinzu, d.h. $T = T'$.
ZEIGE: falsch

1.1 $\langle 2 \rangle 1.$ Es existiert ein endlicher Ablauf σ von Σ' der nicht in *SSPEC* liegt.
BEWEIS: Folgt aus der Voraussetzung, daß Σ' *SSPEC* verletzt, und der Tatsache, daß *SSPEC* eine Sicherheitsspezifikation ist. □

1.2 $\langle 2 \rangle 2.$ Alle Transitionen von σ sind auch Transitionen von Σ.
BEWEIS: Folgt aus Schritt $\langle 2 \rangle 1$ und der Annahme von Schritt $\langle 1 \rangle 1$. □

1.3 ⟨2⟩3. σ ist ein Ablauf von Σ.
BEWEIS: Folgt aus Schritt ⟨2⟩2. □
1.4 ⟨2⟩4. Σ verletzt *SSPEC*.
BEWEIS: Folgt aus Schritt ⟨2⟩3. □
1.5 ⟨2⟩5. Q.E.D.
BEWEIS: Schritt ⟨2⟩4 widerspricht der Voraussetzung, daß Σ *SSPEC* erfüllt. □
2 ⟨1⟩2. Q.E.D.
BEWEIS: Folgt indirekt aus Schritt ⟨1⟩1. □

Wir untersuchen nun Kriterien, mit denen man im voraus abschätzen kann, ob das durch F verursachte Fehlverhalten tolerierbar ist oder nicht.

Gemäß Lemma 3.22 lassen sich Verletzungen von Sicherheitsspezifikationen auf die durch F hinzugefügten Transitionen zurückführen. Die hinzugefügten Transitionen müssen jedoch nicht unbedingt “schlechte” Transitionen im Sinne von Lemma 3.20 sein. Wir wissen nur: Für jeden Ablauf, der *SSPEC* verletzt, gibt es eine Stelle, an der erstmals eine “schlechte” Transition auftritt. Wir zeigen nun, daß auf dem Weg zu diesem Punkt eine nicht-erreichbare Programmtransition von Σ_1 aufgetreten sein muß. Dies gilt jedoch nur unter der Voraussetzung, daß F sicherheitskonservativ ist.

3.23 Definition (nicht-erreichbare Transition) Sei $\Sigma = (C, I, T, A)$ ein Programm. Eine Transition $t \in T$ ist eine *nicht-erreichbare Transition* von Σ falls für alle Abläufe σ von Σ gilt: t kommt nicht in σ vor.

3.24 Definition (minimal verletzender Präfix) Sei $\Sigma = (C, I, T, A)$ ein Programm, *SSPEC* eine Sicherheitsspezifikation und σ ein Ablauf von Σ, der nicht in *SSPEC* liegt. Es gelte $\forall x \in I.\ x$ bewahrt *SSPEC*. Der Präfix $\alpha \cdot s$ von σ heißt *minimal verletzender Präfix von* σ, falls gilt:

- α bewahrt *SSPEC*, aber
- $\alpha \cdot s$ bewahrt nicht *SSPEC*.

Die Existenz eines minimal verletzenden Präfixes folgt aus Propositionen 3.10 und 3.11. Daß α nicht leer ist, folgt aus der Voraussetzung, daß alle initialen Zustände *SSPEC* bewahren.

Das folgende Lemma zeigt: In jedem minimal verletzenden Präfix kommt eine nicht-erreichbare Programmtransition vor.

3.25 Lemma Sei F ein sicherheitskonservatives Fehlermodell, *SSPEC* eine Sicherheitsspezifikation, $\Sigma_1 = (C_1, I_1, T_1, A_1)$ und $\Sigma_2 = (C_2, I_2, T_2, A_2)$ Programme, Σ_2

die F-tolerante Version von Σ_1 für *SSPEC*, σ ein Ablauf von $F(\Sigma_1)$, der *SSPEC* verletzt. Dann gilt: Im minimal verletzenden Präfix $\alpha \cdot s$ von σ kommt eine nicht-erreichbare Transition $t \in T_1$ vor.

Beweis

ANNAHME:
1. $\Sigma_1 = (C_1, I_1, T_1, A_1)$ und $\Sigma_2 = (C_2, I_2, T_2, A_2)$ sind Programme,
2. F ist ein sicherheitskonservatives Fehlermodell,
3. *SSPEC* ist eine Sicherheitsspezifikation,
4. Σ_2 ist eine F-tolerante Version von Σ_1 für *SSPEC*,
5. σ ist ein Ablauf von Σ_1 mit $\sigma \notin SSPEC$, und
6. $\alpha \cdot s$ ist der minimal verletzende Präfix von σ.

ZEIGE: $\exists t.\ t$ kommt in $\alpha \cdot s$ vor und t ist eine nicht-erreichbare Transition von Σ_1.

BEWEISIDEE: Wir nehmen das Gegenteil an und leiten einen Widerspruch her. Wenn nämlich alle Transitionen im minimal verletzenden Präfix erreichbar wären, dann könnte man wegen der Sicherheitskonservativität diesen Präfix komplett in $F(\Sigma_2)$ nachspielen. Dies widerspricht der Annahme, daß Σ_2 die fehlertolerante Version von Σ_1 ist.

1 $\langle 1 \rangle 1$. ANNAHME: Für alle Transitionen t, die in $\alpha \cdot s$ vorkommen, gilt: $t \in T_1$ impliziert t ist eine erreichbare Transition von Σ_1.
ZEIGE: falsch

1.1 $\langle 2 \rangle 1$. Für alle Transitionen t, die in $\alpha \cdot s$ vorkommen, gilt: $t \in F(T_2)$.

1.1.1 $\langle 3 \rangle 1$. ANNAHME: $t \in T_1$
ZEIGE: Q.E.D.

1.1.1.1 $\langle 4 \rangle 1$. t ist eine erreichbare Transition von Σ_1.
BEWEIS: Folgt aus aus der Annahme aus Schritt $\langle 1 \rangle 1$. □

1.1.1.2 $\langle 4 \rangle 2$. $t \in T_2$
BEWEIS: Folgt aus Schritt $\langle 4 \rangle 1$ und der Voraussetzung, daß $\Sigma_2 \geq \Sigma_1$. □

1.1.1.3 $\langle 4 \rangle 3$. Q.E.D.
BEWEIS: Schritt $\langle 4 \rangle 2$ impliziert $t \in F(T_2)$. □

1.1.2 $\langle 3 \rangle 2$. ANNAHME: $t \in F(T_1) \setminus T_1$
ZEIGE: Q.E.D.
BEWEIS: Da t eine durch F hinzugefügte Transition ist, folgt aus der Sicherheitskonservativität von F, daß t auch zu T_2 hinzugefügt wird. Demnach gilt: $t \in F(T_2)$. □

1.1.3 $\langle 3 \rangle 3$. Q.E.D.
BEWEIS: Schritte $\langle 3 \rangle 1$ und $\langle 3 \rangle 2$ decken alle Fälle ab. □

1.2 $\langle 2 \rangle 2$. $\alpha \cdot s$ ist ein Ablauf von $F(\Sigma_2)$.
BEWEIS: Folgt aus Schritt $\langle 2 \rangle 1$. □

1.3 $\langle 2 \rangle 3$. $F(\Sigma_2)$ verletzt *SSPEC*.

BEWEIS: Folgt aus Schritt $\langle 2 \rangle 2$ und der Annahme, daß der Ablauf $\alpha \cdot s \notin$ *SSPEC*. □

1.4 $\langle 2 \rangle 4$. Q.E.D.

BEWEIS: Schritt $\langle 2 \rangle 3$ ist ein Widerspruch zur Voraussetzung, daß Σ_2 eine F-tolerante Version von Σ_1 für *SSPEC* ist. □

2 $\langle 1 \rangle 2$. Q.E.D.

BEWEIS: Folgt indirekt aus Schritt $\langle 1 \rangle 1$. □

Gemäß Lemma 3.25 kann man also alle Abläufe σ, die eine Sicherheitsspezifikation verletzen, aufgrund der Art des in σ liegenden minimal verletzenden Präfixes $\alpha \cdot s$ in zwei Klassen einteilen:

1. Die Menge aller Abläufe, in denen in α ausschließlich erreichbare Transitionen von Σ_1 und Fehlertransitionen vorkommen.

2. Die Menge aller Abläufe, in denen zusätzlich noch nicht-erreichbare Transitionen von Σ_1 auftreten.

Lemma 3.25 besagt, daß — wenn überhaupt — nur die Fehlermodelle F toleriert werden können, die zu Abläufen der zweiten Art führen. Hat F zur Folge, daß auch nur *ein* Ablauf der ersten Art zum Verhalten von $F(\Sigma_1)$ hinzugefügt wird, dann ist F nicht tolerierbar bezüglich *SSPEC*. Der Grund hierfür liegt auf der Hand: Wenn die Verletzung von *SSPEC* allein durch normale Programmtransitionen und Fehlertransitionen erreicht werden kann, dann ist dies auch in jedem Programm Σ_2 möglich, welches Σ_1 erweitert, da Σ_2 dieselben Transitionen aufweist.

3.6.3 Speicherredundanz

Welche Grundmechanismen spielen eine Rolle, wenn man fehlertolerante Versionen bezüglich einer Sicherheitsspezifikation konstruieren will? Eine erste Antwort gibt die Arora/Kulkarni-Theorie: Man benötigt *Detektoren*. Detektoren sind jedoch abstrakte Gebilde, die über ihre Schnittstelle beschrieben werden. In diesem Abschnitt werfen wir einen konkreten Blick auf Detektoren und untersuchen, wie Detektoren prinzipiell funktionieren.

Detektoren verhindern die Ausführung einer Transition, wenn dieser Schritt zu der Verletzung der Sicherheitsspezifikation führen würde. Die Fusionsabgeschlossenheit macht es möglich, diese "schlechten" Transitionen zu identifizieren. Die Wirkung von Detektoren entspricht demnach einem Löschen aller schlechten Transitionen. Wenn dies nicht möglich ist (weil z.B. die schlechten Transitionen durch das Fehlermodell eingefügt wurden), werden wenigstens die nichterreichbaren Transitionen weggeschnitten, die auf dem Weg zu einer solchen

schlechten Transition liegen. Offensichtlich notwendig hierfür sind also nicht-erreichbare Zustände.

3.26 Definition (nicht-erreichbaren Zustand) Sei $\Sigma = (C, I, T, A)$ ein Programm. Ein Zustand $s \in C$ ist ein *nicht-erreichbarer Zustand* von Σ falls für alle Abläufe $\sigma = s_1, s_2, \ldots$ von Σ gilt $s \neq s_i$ für alle i.

Nicht-erreichbare Zustände sind fundamental zur Bewahrung von Sicherheitsspezifikationen. Ohne derartige Zustände kann man nicht fehlertolerant sein. Nicht-erreichbare Zustände sind darum eine gute Möglichkeit, um den Begriff der Speicherredundanz (*redundancy in space*) zu definieren.

3.27 Definition (Speicherredundanz) Ein Programm Σ hat *Speicherredundanz* (oder: ist *speicherredundant*), falls es nicht-erreichbare Zustände hat.

3.28 Theorem (Notwendigkeit von Speicherredundanz) Seien Σ_1 und Σ_2 Programme, F ein sicherheitskonservatives Fehlermodell und *SSPEC* eine Sicherheitsspezifikation. Falls Σ_2 die F-tolerante Version von Σ_1 für *SSPEC* ist, dann hat Σ_2 Speicherredundanz.

Beweis

ANNAHME:
1. $\Sigma_1 = (C_1, I_1, T_1, A_1)$ und $\Sigma_2 = (C_2, I_2, T_2, A_2)$ sind Programme,
2. F ist ein sicherheitskonservatives Fehlermodell,
3. *SSPEC* ist eine Sicherheitsspezifikation,
4. Σ_2 ist eine F-tolerante Version von Σ_1 für *SSPEC*, und
5. $\forall x \in I_1$ gilt, daß x *SSPEC* bewahrt.

ZEIGE: Σ_2 hat Speicherredundanz.

BEWEISIDEE: Der Beweis ist konstruktiv: Wir nehmen einen Ablauf σ, der *SSPEC* verletzt. Gemäß Lemma 3.25 kommt darin eine nicht-erreichbare Programmtransition von Σ_1 vor. Jetzt betrachten wir den maximalen Präfix $\alpha \cdot s$ von σ, in dem keine solche nicht-erreichbare Transition vorkommt. Da dieser Präfix maximal ist, muß s der Ausgangspunkt einer nicht-erreichbaren Transition von Σ_1 sein. Das bedeutet, s ist ein nicht-erreichbarer Zustand von Σ_1. Jetzt ist es einfach zu zeigen, daß s auch ein nicht-erreichbarer Zustand von Σ_2 ist.

1 ⟨1⟩1. Für jeden Ablauf σ von $F(\Sigma_1)$, der *SSPEC* verletzt, gibt es einen Präfix β, in dem eine nicht-erreichbare Transition $t = (s, s')$ von Σ_1 vorkommt.
BEWEIS: Folgt aus Lemma 3.25. □

2 ⟨1⟩2. Betrachte den maximalen Präfix $\alpha \cdot s$ von β, in dem t nicht vorkommt, d.h. $\beta = \alpha \cdot s \cdot s' \cdot \gamma$ für α maximal. Ohne Beschränkung der Allgemeinheit gilt: In $\alpha \cdot s$ kommen keine nicht-erreichbaren Transitionen von Σ_1 vor.

BEWEIS: Falls in $\alpha \cdot s$ doch nicht-erreichbare Transitionen vorkommen, definiere Transition t aus Schritt $\langle 1 \rangle 1$ einfach als die erste in $\alpha \cdot s$ vorkommende solche Transition. $\square$

3 $\langle 1 \rangle 3$. $\alpha \cdot s$ der Präfix eines Ablaufs von $F(\Sigma_2)$.

3.1 $\langle 2 \rangle 1$. Für alle Transitionen u in $\alpha \cdot s$ gilt: u ist eine erreichbare Transition von Σ_1 oder u ist eine durch F hinzugefügte Transitionen.

BEWEIS: Folgt aus Schritt $\langle 1 \rangle 2$ und der Sicherheitskonservativität von F. $\square$

3.2 $\langle 2 \rangle 2$. Für alle Transitionen u in $\alpha \cdot s$ gilt: $u \in F(T_2)$.

3.2.1 $\langle 3 \rangle 1$. ANNAHME: u ist eine erreichbare Transition von Σ_1.
ZEIGE: Q.E.D.

BEWEIS: Wegen $\Sigma_2 \geq \Sigma_1$ folgt, daß $u \in T_2$ und somit auch $u \in F(T_2)$. $\square$

3.2.2 $\langle 3 \rangle 2$. ANNAHME: u ist eine durch F hinzugefügte Transition.
ZEIGE: Q.E.D.

BEWEIS: Wegen der Sicherheitskonservativität von F wird u auch zu T_2 hinzugefügt. $\square$

3.2.3 $\langle 3 \rangle 3$. Q.E.D.

BEWEIS: Wegen Schritt $\langle 2 \rangle 1$ decken Schritte $\langle 3 \rangle 1$ und $\langle 3 \rangle 2$ alle Fälle ab. $\square$

3.3 $\langle 2 \rangle 3$. Q.E.D.

BEWEIS: Folgt direkt aus Schritt $\langle 2 \rangle 2$. $\square$

4 $\langle 1 \rangle 4$. s ist ein nicht-erreichbarer Zustand von Σ_1.

BEWEIS: Wenn $t = (s, s')$ eine nicht-erreichbare Transition von Σ_1 ist (Schritt $\langle 1 \rangle 1$), muß s ein nicht-erreichbarer Zustand von Σ_1 sein. $\square$

5 $\langle 1 \rangle 5$. s ist ein nicht-erreichbarer Zustand von Σ_2.

5.1 $\langle 2 \rangle 1$. ANNAHME: s ist ein erreichbarer Zustand von Σ_2.
ZEIGE: falsch

5.1.1 $\langle 3 \rangle 1$. Es gibt einen Ablauf ρ von Σ_2, in dem s vorkommt.

BEWEIS: Folgt aus der Annahme aus Schritt $\langle 2 \rangle 1$. $\square$

5.1.2 $\langle 3 \rangle 2$. ρ ist ein Ablauf von Σ_1.

BEWEIS: Folgt aus Schritt $\langle 3 \rangle 1$ und der Voraussetzung, daß $\Sigma_2 \geq \Sigma_1$. $\square$

5.1.3 $\langle 3 \rangle 3$. Q.E.D.

BEWEIS: Aus Schritt $\langle 3 \rangle 2$ folgt, daß s ein erreichbarer Zustand von Σ_1 ist. Dies ist ein Widerspruch zu Schritt $\langle 1 \rangle 4$. $\square$

5.2 $\langle 2 \rangle 2$. Q.E.D.

BEWEIS: Folgt indirekt aus Schritt $\langle 2 \rangle 1$. $\square$

6 $\langle 1 \rangle 6$. Q.E.D.

BEWEIS: Folgt direkt aus Schritt $\langle 1 \rangle 5$ und Definition 3.27. $\square$

3.7 Fehlertoleranzmechanismen für Lebendigkeit

In diesem Abschnitt wenden wir uns Lebendigkeitsspezifikationen zu und untersuchen Methoden, wie man Lebendigkeit erreichen kann, auch wenn Fehler auftreten.

In der Literatur hat man sich bisher hauptsächlich mit Sicherheitseigenschaften beschäftigt. Der Grund hierfür liegt sicherlich zum Teil darin, daß Lebendigkeitseigenschaften etwas schwieriger zu handhaben sind als Sicherheitseigenschaften [151]. Im Rahmen von Fehlertoleranzuntersuchungen werden Lebendigkeitseigenschaften oft unter Zuhilfenahme von Sicherheitseigenschaften abgehandelt. Dazu wird angenommen, daß das System lebendig ist, wenn ein Prädikat φ auf dem Zustand gilt. Lebendigkeit ist also garantiert, wenn das System die Spezifikation $\Box\varphi$ erfüllt, eine Sicherheitseigenschaft.

Wir wollen diese Einschränkung nicht machen, d.h. wir betrachten allgemeine Lebendigkeitseigenschaften. Wir stellen zunächst in Abschnitt 3.7.1 einen geeigneten Konservativitätsbegriff für Lebendigkeit vor und untersuchen dann Kriterien zur Tolerierbarkeit von gegebenen Fehlermodellen (Abschnitt 3.7.2). Anschließend definieren wir in Abschnitt 3.7.3 den Begriff der *Zeitredundanz* und weisen nach, daß Zeitredundanz notwendig ist für das Erreichen von Lebendigkeit.

3.7.1 Lebendigkeitskonservative Fehlermodelle

Betrachten wir das Programm Σ aus Abbildung 3.9 (links). Die Lebendigkeitsspezifikation *LSPEC*, die es erfüllen soll, ist $\Box\Diamond c$. Da die Lebendigkeitsannahme von Σ schwache Fairneß ist, erfüllt Σ offensichtlich *LSPEC*.

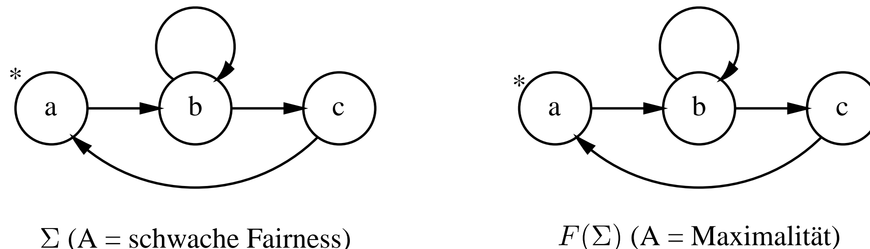


Abbildung 3.9: Abschwächen der Lebendigkeit genügt, um Lebendigkeitsspezifikationen zu verletzen.

Wir können nun ein Fehlermodell F definieren, welches keine Transitionen hinzufügt, sondern lediglich die Lebendigkeitsannahme auf Maximalität abschwächt.

Nun wird ein Ablauf möglich, der einen unendlichen Suffix aus b's hat. Demnach verletzt $F(\Sigma)$ nun *LSPEC*. Ein Fehlermodell kann also allein durch die Abschwächung der Lebendigkeitsannahme dafür sorgen, daß eine Lebendigkeitsspezifikation verletzt wird.

Unserer Erfahrung nach erwartet man von einem Fehlermodell, daß es sich bezüglich der Lebendigkeitsannahmen verschiedener Programme gleich verhält. Es ist also sinnvoll anzunehmen, daß F die Lebendigkeitsannahmen der beiden involvierten Programme in gleicher Weise abschwächt. Dies führt zum Konzept der *Lebendigkeitskonservativität*.

3.29 Definition (lebendigkeitskonservatives Fehlermodell) Seien $\Sigma_1 = (C_1, I_1, T_1, A_1)$ und $\Sigma_2 = (C_2, I_2, T_2, A_2)$ Programme mit $\Sigma_2 \geq \Sigma_1$. Ein Fehlermodell F heißt *lebendigkeitskonservativ* falls gilt: $F(A_1) = F(A_2)$.

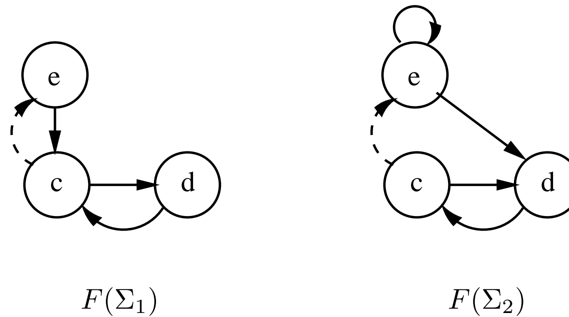


Abbildung 3.10: Beispiel für ein lebendigkeitskonservatives Fehlermodell.

Betrachten wir Abbildung 3.10 und sei $LSPEC = \Box\Diamond d$. Wenn man für das Programm Σ_1 auf der linken Seite auch im Fehlerfall starke Fairneß annimmt, dann erfüllt selbst $F(\Sigma_1)$ *LSPEC*. Wenn F allerdings die Lebendigkeitsannahme auf schwache Fairneß abschwächt, gilt *LSPEC* nicht mehr für $F(\Sigma_1)$. Es ist allerdings möglich, eine F-tolerante Version Σ_2 zu konstruieren, die auch im Fehlerfall *LSPEC* erfüllt. Dies ist auf der rechten Seite der Abbildung dargestellt. Man muß allerdings davon ausgehen, daß F lebendigkeitskonservativ ist, d.h. daß auch für $F(\Sigma_2)$ mindestens schwache Fairneß gilt. Andernfalls verletzt $F(\Sigma_2)$ *LSPEC*.

Nach Definition 3.16 gilt wegen $\Sigma_2 \geq \Sigma_1$ bereits $A_1 = A_2$. Ein lebendigkeitskonservatives Fehlermodell sorgt also dafür, daß auch die fehlerbehafteten Versionen von Σ_1 und Σ_2 dieselbe (wenn auch möglicherweise schwächere) Lebendigkeitsannahme besitzen.

Wenn man Lebendigkeitskonservativität annimmt, ergeben sich interessante Folgerungen über das Verhalten von fehlertoleranten Versionen. Das folgende Lemma besagt: Wenn Σ_2 die F-tolerante Version von Σ_1 für *LSPEC* ist, dann kann

man bestimmte Abläufe von $F(\Sigma_1)$ nicht beliebig lange mittels $F(\Sigma_2)$ "nachspielen".

3.30 Lemma Seien Σ_1 und Σ_2 Programme, F ein lebendigkeitskonservatives Fehlermodell, *LSPEC* eine Lebendigkeitsspezifikation, und Σ_2 die F-tolerante Version von Σ_1 für *LSPEC*. Dann gilt: Für jeden Ablauf σ von $F(\Sigma_1)$, der *LSPEC* verletzt, gibt es einen maximalen Präfix α, der in $F(\Sigma_2)$ fortgesetzt werden kann.

Beweis

ANNAHME:
1. $\Sigma_1 = (C_1, I_1, T_1, A_1)$ und $\Sigma_2 = (C_2, I_2, T_2, A_2)$ sind Programme,
2. F ist ein lebendigkeitskonservatives Fehlermodell,
3. *LSPEC* ist eine Lebendigkeitsspezifikation,
4. Σ_2 ist die F-tolerante Version von Σ_1 für *LSPEC*.
5. σ ist ein Ablauf von $F(\Sigma_1)$ mit $\sigma \notin LSPEC$.

ZEIGE: Es existiert ein Präfix α von σ für den gilt:
1. es existiert ein β so daß $\alpha \cdot \beta$ ein Ablauf von $F(\Sigma_2)$ ist, und
2. für alle Präfixe α' von σ, die länger als α sind, gilt: für alle β': $\alpha' \cdot \beta'$ ist kein Ablauf von $F(\Sigma_2)$.

BEWEISIDEE: Der Beweis ist indirekt. Unter der Annahme des Gegenteils kann man zeigen, daß jeder Zustandsübergang in σ auch ein Zustandsübergang von $F(\Sigma_2)$ sein muß. Darum kann σ beliebig lange, d.h. auch unendlich lange, mittels $F(\Sigma_2)$ nachgespielt werden (hier geht ein, daß beide Programme dieselbe Lebendigkeitsannahme besitzen). Das bedeutet aber, daß $F(\Sigma_2)$ *LSPEC* verletzen muß, ein Widerspruch.

1 $\langle 1 \rangle 1$. ANNAHME: Alle Präfixe von σ können in $F(\Sigma_2)$ fortgesetzt werden, d.h. für alle Präfixe α von σ gibt es ein β so daß $\alpha \cdot \beta$ ein Ablauf von $F(\Sigma_2)$ ist.
ZEIGE: falsch

1.1 $\langle 2 \rangle 1$. Jeder Zustandsübergang t in σ ist ein Zustandsübergang von $F(\Sigma_2)$, d.h. $t \in F(T_2)$.
BEWEIS: Für jeden Zustandsübergang t in σ gibt es einen Präfix α von σ, in dem t enthalten ist. Wenn α durch $F(\Sigma_2)$ fortgesetzt werden kann, muß $t \in F(T_2)$ gelten. □

1.2 $\langle 2 \rangle 2$. σ ist ein Ablauf von $F(\Sigma_2)$.
BEWEIS: Folgt aus Schritt $\langle 2 \rangle 1$ und der Voraussetzung, daß $F(A_1) = F(A_2)$ (Lebendigkeitskonservativität von F). □

1.3 $\langle 2 \rangle 3$. Q.E.D.
BEWEIS: Da $\sigma \notin LSPEC$ (nach Voraussetzung) verletzt $F(\Sigma_2)$ wegen der Feststellung in Schritt $\langle 2 \rangle 2$ *LSPEC*. Dies ist ein Widerspruch zur Voraussetzung, daß $F(\Sigma_2)$ eine F-tolerante Version für *LSPEC* ist. □

2 ⟨1⟩2. Q.E.D.
BEWEIS: Die Konklusion des Lemmas folgt indirekt aus Schritt ⟨1⟩1. □

Lemma 3.30 ist fundamental für die weiteren Ergebnisse dieses Abschnitts. Es stellt fest, daß unter den gegebenen Voraussetzungen die Abläufe der Programme $F(\Sigma_1)$ und $F(\Sigma_2)$ aufgrund einer "fehlenden" Transition in T_2 auseinandergehen müssen. Das wird offensichtlich an folgender Aufgabe: Gegeben ein Fehlermodell F, welches keinerlei Transitionen hinzufügt sondern ausschließlich die Lebendigkeitsannahme eines Programmes abschwächt. Für Programme, bei denen dies zur Verletzung einer gegebenen Lebendigkeitsspezifikation *LSPEC* führt, finde eine F-tolerante Version! Es stellt sich heraus, daß dies nicht möglich ist.

3.31 Proposition Gegeben eine Lebendigkeitsspezifikation *LSPEC*, ein Programm Σ_1, welches *LSPEC* erfüllt, und ein Fehlermodell F, unter welchen Σ_1 *LSPEC* verletzt. F fügt keinerlei Transitionen hinzu, d.h. die Verletzung von *LSPEC* wird ausschließlich durch die Abschwächung der Lebendigkeitsannahme erreicht. Dann gilt: Es gibt keine F-tolerante Version Σ_2 von Σ_1 für *LSPEC*.

Beweis

ANNAHME:
1. *LSPEC* ist eine Lebendigkeitsspezifikation,
2. Σ_1 ist ein Programm, welches *LSPEC* erfüllt,
3. F ist ein lebendigkeitskonservatives Fehlermodell, welches keinerlei Transitionen hinzufügt, und
4. $F(\Sigma_1)$ verletzt *LSPEC*.

ZEIGE: Es existiert keine F-tolerante Version Σ_2 von Σ_1 für *LSPEC*.

BEWEISIDEE: Wir nehmen an, es gäbe eine F-tolerante Version und leiten einen Widerspruch her. Wenn es nämlich ein solches Programm Σ_2 gäbe, dann gibt es nach Lemma 3.30 einen maximalen Nachspielpräfix α für jeden Ablauf σ von $F(\Sigma_1)$, der *LSPEC* verletzt. Unter den gegebenen Annahmen kann man zeigen, daß die Transition, die im Anschluß an α genommen wird, auch eine Transition von Σ_2 ist. Dies ist ein Widerspruch zur Annahme, daß α maximal ist.

3.7.2 Tolerierbare und nicht tolerierbare Fehler

Verletzungen der Lebendigkeit geschehen im Unendlichen, d.h. ein Ablauf σ, der eine Lebendigkeitsspezifikation *LSPEC* verletzt, besitzt einen "unerwünschten" unendlichen Suffix. Wie muß ein solcher Suffix aussehen, damit das ihn verursachende Fehlermodell toleriert werden kann? Eine hinreichende Bedingung

ergibt sich aus dem folgenden Lemma: In einem derartigen Ablauf muß eine nicht-erreichbare Transition vorkommen.

Eine Aussage über die Gestalt von F gab bereits Proposition 3.31: F muß mindestens eine Transition hinzufügen. Es ist darum sinnvoll, Sicherheitskonservativität von F anzunehmen.

3.32 Lemma Seien Σ_1 und Σ_2 Programme, F ein sicherheits- und lebendigkeitskonservatives Fehlermodell, *LSPEC* eine Lebendigkeitsspezifikation, und Σ_2 die F-tolerante Version von Σ_1 für *LSPEC*. Dann gilt: Für alle Abläufe σ von $F(\Sigma_1)$, die *LSPEC* verletzen, existiert eine nicht-erreichbare Transition $t \in T_1$, so daß t in σ vorkommt.

Beweis

ANNAHME:
1. $\Sigma_1 = (C_1, I_1, T_1, A_1)$ und $\Sigma_2 = (C_2, I_2, T_2, A_2)$ sind Programme,
2. F ist ein sicherheits- und lebendigkeitskonservatives Fehlermodell,
3. *LSPEC* ist eine Lebendigkeitsspezifikation,
4. Σ_2 ist die F-tolerante Version von Σ_1 für *LSPEC*.
5. σ ist ein Ablauf von $F(\Sigma_1)$ mit $\sigma \notin LSPEC$.

ZEIGE: Es existiert eine Transition $t \in T_1$, für die gilt: t ist eine nicht-erreichbare Transition von Σ_1 und t kommt in σ vor.

BEWEISIDEE: Der Beweis ist indirekt. Wir nehmen an, alle derartigen Transitionen t sind erreichbar. Dann sind dies aber auch alles Transitionen von $F(\Sigma_2)$ und man kann den Ablauf σ komplett in $F(\Sigma_2)$ nachspielen, ein Widerspruch zur Tatsache, daß $F(\Sigma_2)$ *LSPEC* erfüllt.

1 $\langle 1 \rangle 1$. ANNAHME: Für alle Transitionen t, die in σ vorkommen, gilt: $t \in T_1$ impliziert t ist erreichbar.
ZEIGE: falsch

1.1 $\langle 2 \rangle 1$. Für alle Transitionen t, die in σ vorkommen, gilt: $t \in F(T_2)$.

1.1.1 $\langle 3 \rangle 1$. ANNAHME: $t \in T_1$
ZEIGE: Q.E.D.
BEWEIS: Wegen der Annahme aus Schritt $\langle 1 \rangle 1$ ist t eine erreichbare Transition von Σ_1. Da $\Sigma_2 \geq \Sigma_1$ nach Voraussetzung, ist t auch eine erreichbare Transition von Σ_2. Dies impliziert, daß $t \in T_2$, und folglich $t \in F(T_2)$. □

1.1.2 $\langle 3 \rangle 2$. ANNAHME: $t \in F(T_1) \setminus T_1$
ZEIGE: Q.E.D.
BEWEIS: Da F sicherheitskonservativ ist und $\Sigma_2 \geq \Sigma_1$, gilt $t \in F(T_2)$. □

1.1.3 $\langle 3 \rangle 3$. Q.E.D.
BEWEIS: Die Annahmen der Schritte $\langle 3 \rangle 1$ und $\langle 3 \rangle 2$ decken alle Fälle ab. □

1.2 $\langle 2 \rangle 2$. σ ist ein Ablauf von $F(\Sigma_2)$.

BEWEIS: Folgt aus Schritt $\langle 2 \rangle 1$ und der Lebendigkeitskonservativität von F. □

1.3 $\langle 2 \rangle 3$. $F(\Sigma_2)$ verletzt $LSPEC$.

BEWEIS: Folgt aus Schritt $\langle 2 \rangle 2$ und der Voraussetzung, daß $\sigma \notin LSPEC$. □

1.4 $\langle 2 \rangle 4$. Q.E.D.

BEWEIS: Schritt $\langle 2 \rangle 3$ ist ein Widerspruch zur Voraussetzung, daß $F(\Sigma_2)$ $LSPEC$ erfüllt. □

2 $\langle 1 \rangle 2$. Q.E.D.

BEWEIS: Die Konklusion des Lemmas folgt indirekt aus Schritt $\langle 1 \rangle 1$. □

Gemäß Lemma 3.32 kann man die Menge aller Abläufe, die $LSPEC$ verletzen, in zwei Klassen einteilen:

1. Die Klasse aller Abläufe, in denen ausschließlich Fehlertransitionen (d.h. Transitionen aus $F(T_1) \setminus T_1$) und erreichbare Transitionen von Σ_1 vorkommen.

2. Die Klasse aller Abläufe, in denen zusätzlich auch nicht-erreichbare Transitionen von Σ_1 vorkommen.

Lemma 3.32 besagt: Hat ein System $F(\Sigma_1)$ einen Ablauf, der zur ersten Kategorie gehört, dann gibt es keine fehlertolerante Version von Σ_1 für $LSPEC$. In diesen Fällen ist also die Kontrolle über das System vollständig zusammengebrochen. Sind alle Abläufe von $F(\Sigma_1)$ in der zweiten Kategorie, besteht immerhin Hoffnung, daß eine fehlertolerante Version existiert. Die Frage nach der Existenz einer fehlertoleranten Version ist zwar entscheidbar [108], uns ist aber kein einfaches Kriterium bekannt, nachdem diese Frage beantwortet werden kann.

3.7.3 Zeitredundanz

Welche Grundmechanismen spielen eine Rolle bei der Konstruktion fehlertoleranter Versionen, wenn man Lebendigkeitsspezifikationen erfüllen will? Eine erste Antwort gibt bereits die Arora/Kulkarni-Theorie: Man benötigt Korrektoren. Korrektoren sind jedoch abstrakte Gebilde, die allein über ihre Schnittstelle definiert sind. In diesem Abschnitt interessieren wir uns für die Mechanismen auf einer niedrigeren Abstraktionsschicht. Wir fragen gewissermaßen: Warum funktionieren Korrektoren?

Die Untersuchung fehlertoleranter Systeme, die Korrektoren enthalten, führen zu folgender Beobachtung: Korrektoren führen eine Berechnung direkt oder indirekt in einen Zustand zurück, von dem aus ein Programm wieder gemäß seiner Spezifikation funktioniert. Es werden also Transitionen eingebaut, die im ursprünglichen

fehler-intoleranten Programm nicht vorhanden waren. Diese “neuen” Transitionen müssen insbesondere *nicht-erreichbare Transitionen* gemäß Definition 3.23 sein. Sie eignen sich darum gut für die Definition des Begriffs der Zeitredundanz.

3.33 Definition (Zeitredundanz) Ein Programm Σ hat *Zeitredundanz* (oder: ist *zeitredundant*), falls es nicht-erreichbare Transitionen hat.

3.34 Theorem (Notwendigkeit von Zeitredundanz) Seien Σ_1 und Σ_2 Programme, F ein sicherheits- und lebendigkeitskonservatives Fehlermodell und *LSPEC* eine Lebendigkeitsspezifikation. Falls Σ_2 die F-tolerante Version von Σ_1 für *LSPEC* ist, dann hat Σ_2 Zeitredundanz.

Beweis

ANNAHME:
1. $\Sigma_1 = (C_1, I_1, T_1, A_1)$ und $\Sigma_2 = (C_2, I_2, T_2, A_2)$ sind Programme,
2. F ist ein sicherheits- und lebendigkeitskonservatives Fehlermodell,
3. *LSPEC* ist eine Lebendigkeitsspezifikation,
4. Σ_2 ist die F-tolerante Version von Σ_1 für *LSPEC*.

ZEIGE: Σ_2 hat Zeitredundanz.

BEWEISIDEE: Wir nehmen einen Ablauf σ von $F(\Sigma_1)$, der *LSPEC* verletzt. Lemma 3.30 garantiert, daß man σ nur maximal lange mittels $F(\Sigma_2)$ nachspielen kann. Die Transition t, die daraufhin durch $F(\Sigma_2)$ genommen wird, ist eine nicht-erreichbare Transition von Σ_2. Wenn sie erreichbar wäre, wäre t auch eine Transition von Σ_1. Das widerspricht der Maximalität des nachgespielten Ablaufs.

1 $\langle 1\rangle 1$. Es existiert ein Ablauf σ von $F(\Sigma_1)$, der nicht in *LSPEC* liegt.
BEWEIS: Folgt aus der Voraussetzung, daß $F(\Sigma_1)$ *LSPEC* verletzt. □

2 $\langle 1\rangle 2$. Es gibt einen maximalen Präfix α von σ, der in $F(\Sigma_2)$ fortgesetzt werden kann.
BEWEIS: Folgt aus Schritt $\langle 1\rangle 1$ und Lemma 3.30. □

3 $\langle 1\rangle 3$. Bezeichne $t_1 = (s, s_1)$ die im Anschluß an α durch $F(\Sigma_1)$ genommene Transition. Dann gilt $t_1 \notin F(T_2)$.
BEWEIS: Folgt aus Schritt $\langle 1\rangle 2$ und der Tatsache, daß beide Programme dieselbe Lebendigkeitsannahme besitzen (hier geht die Lebendigkeitskonservativität ein). □

4 $\langle 1\rangle 4$. Bezeichne $t_2 = (s, s_2)$ die im Anschluß an α durch $F(\Sigma_2)$ genommene Transition. Eine solche Transition existiert, ist ungleich t_1 und es gilt $t_2 \in T_2$.
BEWEIS: Die Existenz folgt aus der Voraussetzung, daß $F(\Sigma_2)$ *LSPEC* erfüllt. Ungleichheit folgt aus Schritt $\langle 1\rangle 3$. $t_2 \in T_2$ folgt aus der Sicherheitskonservativität von F (Rückrichtung). □

5 $\langle 1\rangle 5$. t_1 ist eine nicht-erreichbare Transition von Σ_1.

5.1 $\langle 2 \rangle 1$. ANNAHME: t_1 ist eine erreichbare Transition von Σ_1.
ZEIGE: falsch

5.1.1 $\langle 3 \rangle 1$. t_1 ist eine erreichbare Transition von Σ_2.
BEWEIS: Folgt aus der Annahme von Schritt $\langle 2 \rangle 1$ und der Voraussetzung, daß $\Sigma_2 \geq \Sigma_1$. □

5.1.2 $\langle 3 \rangle 2$. $t_1 \in T_2$
BEWEIS: Folgt aus Schritt $\langle 3 \rangle 1$. □

5.1.3 $\langle 3 \rangle 3$. $t_1 \in F(T_2)$
BEWEIS: Folgt aus Schritt $\langle 3 \rangle 2$ und Definition 3.12. □

5.1.4 $\langle 3 \rangle 4$. Q.E.D.
BEWEIS: Schritt $\langle 3 \rangle 3$ widerspricht Schritt $\langle 1 \rangle 3$. □

5.2 $\langle 2 \rangle 2$. Q.E.D.
BEWEIS: Folgt indirekt aus Schritt $\langle 2 \rangle 1$. □

6 $\langle 1 \rangle 6$. t_2 ist eine nicht-erreichbare Transition von Σ_2.
BEWEIS: Schritt $\langle 1 \rangle 5$ impliziert, daß s (der Startzustand von t_1) ein nicht-erreichbarer Zustand von Σ_1 ist. Da s auch der Startzustand von t_2 ist, muß t_2 eine nicht-erreichbare Transition von Σ_2 sein. □

7 $\langle 1 \rangle 7$. Q.E.D.
BEWEIS: Folgt direkt aus Schritt $\langle 1 \rangle 6$ und Definition 3.33. □

3.8 Beispiele

Die Nützlichkeit der Redundanzdefinitionen soll anhand zweier Beispiele dargelegt werden. Die vorgestellten Definitionen machen es in diesen Fällen sogar möglich, verschiedene Programme bezüglich ihrer Redundanz zu vergleichen und in dieser Beziehung Aussagen über die Optimalität von Fehlertoleranzmechanismen zu machen.

3.8.1 Minimale Redundanz bei TMR

Wir betrachten wieder das in Abschnitt 3.2.3 bereits vorgestellte Programm für *triple modular redundancy*. Der Zustandsraum des fehlertoleranten Programms ist in Abbildung 3.11 dargestellt. Es wird angenommen, daß der von den einzelnen Komponenten x, y und z berechnete Wert entweder 0 oder 1 ist. Bei den Zustandsübergängen wird in der Abbildung zwischen drei Klassen unterschieden:

1. Fehlertransitionen sind wie immer durch gestrichelte Linien dargestellt.

2. Transitionen, die Teil des Programmes sind, welches nur mittels Detektoren

ausgestattet ist (d.h. nur korrekt ist bezüglich der Sicherheitsspezifikation), sind mit durchgezogenen Linien dargestellt.

3. Transitionen, die zusätzlich benötigt werden, um Lebendigkeit zu erreichen, sind mit gepunkteten Linien dargestellt.

Die Korrektheitsspezifikation lautet:

- $SSPEC$ = *out* nimmt nie den Wert einer durch Fehler veränderten Komponente an.
- $LSPEC$ = *out* wird nach endlicher Zeit einen Wert ungleich $\perp$ annehmen.

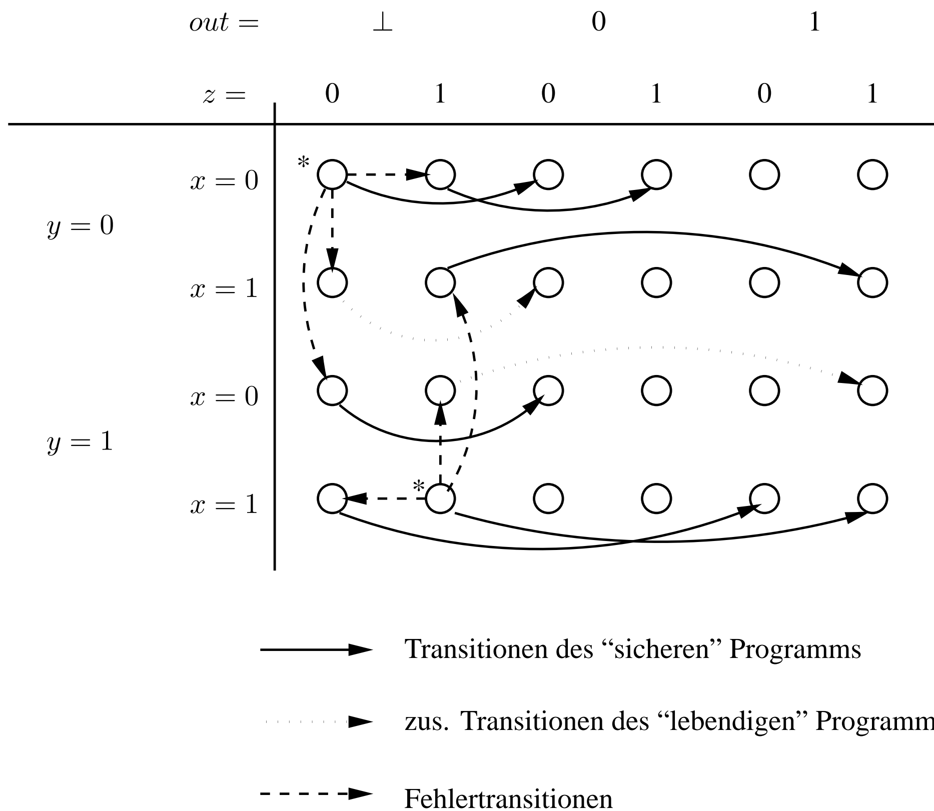


Abbildung 3.11: Zustandsraum des fehlertoleranten Programms für *triple modular redundancy*.

Aus einem der beiden möglichen initialen Zuständen kann ein Fehler den Berechnungswert einer einzelnen Komponente negieren. Wenn y oder z durch den Fehler verändert wurden, dann reicht die Basisfunktionalität aus, um Sicherheit und Lebendigkeit zu erreichen. Wenn jedoch x verändert wird, dann

bleibt das Programm, welches nur aus Detektoren besteht, entweder im Zustand $(x, y, z, out) = (1, 0, 0, \perp)$ oder im Zustand $(x, y, z, out) = (0, 1, 1, \perp)$ stehen. Zwar ist *SSPEC* immer erfüllt, jedoch *LSPEC* nicht. Um in jedem Fall auch *LSPEC* zu erfüllen, müssen die beiden zusätzlichen gepunkteten Transitionen eingefügt werden.

Es ergibt sich die folgende Beobachtung: Bezüglich Zeitredundanz ist dieses Programm optimal. Es kann kein Programm geben, welches F-tolerant bezüglich *SSPEC* und *LSPEC* ist und weniger Transitionen besitzt, da mindestens aus jedem Fehlerzustand ein Pfad herausführen muß.

Wenn man lediglich *SSPEC* erfüllen will, dann ist das Programm auch ohne die beiden zusätzlichen Transitionen nicht optimal bezüglich Speicherredundanz. Man kann nämlich durch Wegstreichen von entweder y oder z ein Programm konstruieren, welches nur fünf statt zehn nicht-erreichbare Zustände besitzt: einen Zustand, um im Fehlerfall "stehenzubleiben", und vier "schlechte" Zustände.

3.8.2 Zeitredundanz bei ARQ-Verfahren

ARQ steht für "*automatic repeat request*" und ist ein Kürzel für Datenübertragungstechniken, die auf einem erneuten Senden einer Nachricht beruhen. Derartige Techniken werden häufig als Anwendungsgebiet von Zeitredundanz genannt.

Wir betrachten einen Übertragungskanal, über den ein Boolescher Wert (0 oder 1) übertragen werden soll. Zur Fehlererkennung wird neben dem eigentlichen Bit c ein weiteres *parity bit* p übertragen. Der zu übertragende Wert liegt an einem Eingang *in* an und soll auf einen Ausgang *out* geschrieben werden. Für die Fehlerkorrektur steht ein Rückkanal a zur Verfügung. Die Sicherheits- und Lebendigkeitsspezifikationen lauten:

- Sicherheit: Wenn ein Wert nach *out* geschrieben wird, dann ist dies der Wert von *in*.
- Lebendigkeit: Nach endlicher Zeit wird ein Wert ungleich $\perp$ nach *out* geschrieben.

Wir beschreiben den Algorithmus zunächst in der Notation der bewachten Anweisungen. Senderseitig wird der Wert von *in* auf den Kanal gegeben, wenn auf diesem noch kein Wert übertragen wurde (dies wird durch einen "Null"-Wert $\perp$ angezeigt). Das *parity bit* ist in diesem Falle gleich dem Wert von *in*. Dies wird so lange gemacht, bis der Rückkanal a anzeigt, daß der Wert korrekt übernommen wurde:

$$\begin{array}{rcl} a = 0 & \rightarrow & c := in, p := in \\ c \neq \perp \wedge a = 1 & \rightarrow & c := \perp \end{array}$$

Das Fehlermodell formalisiert einen Übertragungsfehler in c, d.h. das dort gespeicherte Bit kann einen zufälligen Wert annehmen. Dies wird durch die beiden folgenden Anweisungen modelliert:

$$
\begin{array}{lcl}
c = 0 & \rightarrow & c := 1 \\
c = 1 & \rightarrow & c := 0
\end{array}
$$

Auf der Empfängerseite muß der Rückkanal entsprechend der Fehlererkennung gesetzt werden. Wenn $c = p$, dann kann eine positive Rückmeldung gegeben werden ($a = 1$). Andernfalls hat ein Übertragungsfehler stattgefunden und das wiederholte Senden bleibt weiterhin eingeschaltet ($a = 0$):

$$
\begin{array}{lcl}
c = p \wedge out = \bot & \rightarrow & out := c, a := 1 \\
c \neq p \wedge out = \bot & \rightarrow & a := 0
\end{array}
$$

Abbildung 3.12 zeigt das Programm nochmals im Überblick. Betrachtet man den Zustandsraum des Programmes, ergeben sich insgesamt 72 Zustände. Abbildung 3.13 zeigt alle Zustände für den Eingabewert $in = 0$. Der initiale Zustand ist durch einen Stern gekennzeichnet. Im einzig möglichen Folgezustand besteht die Möglichkeit eines Fehlers (Transition von $(p, a, c) = (0, 0, 0)$ nach $(0, 0, 1)$ und wieder zurück). Annahme von starker Fairneß impliziert, daß diese Fehler nach endlicher Zeit aufhören und irgendwann die Transition zum Ausgangszustand des Fehlers zurück genommen wird. Die Transition von $(p, a, c) = (0, 0, 1)$ nach $(0, 0, 0)$ ist ein Beispiel für eine redundante Transition und modelliert das wiederholte Senden, bis das Resultat korrekt am Empfänger angekommen ist. Anschließend wird empfängerseitig das Ergebnis nach *out* übernommen und der Kanal in den Zustand $\bot$ zurückgesetzt.

variables: $c \in \{0, 1, \bot\}$ **initially** $\bot$ (channel)
$p \in \{0, 1\}$ **initially** 0 (parity)
$a \in \{0, 1\}$ **initially** 0 (arq channel)
$in \in \{0, 1\}$ (input value)
$out \in \{0, 1, \bot\}$ **initially** $\bot$ (output value)
actions:

$$
\begin{array}{lcl}
a = 0 & \rightarrow & c := in, p := in \\
c \neq \bot \wedge a = 1 & \rightarrow & c := \bot \\
c = 0 & \rightarrow & c := 1 \\
c = 1 & \rightarrow & c := 0 \\
c = p \wedge out = \bot & \rightarrow & out := c, a := 1 \\
c \neq p \wedge out = \bot & \rightarrow & a := 0
\end{array}
$$

Abbildung 3.12: ARQ-Programm in der Notation der bewachten Anweisungen.

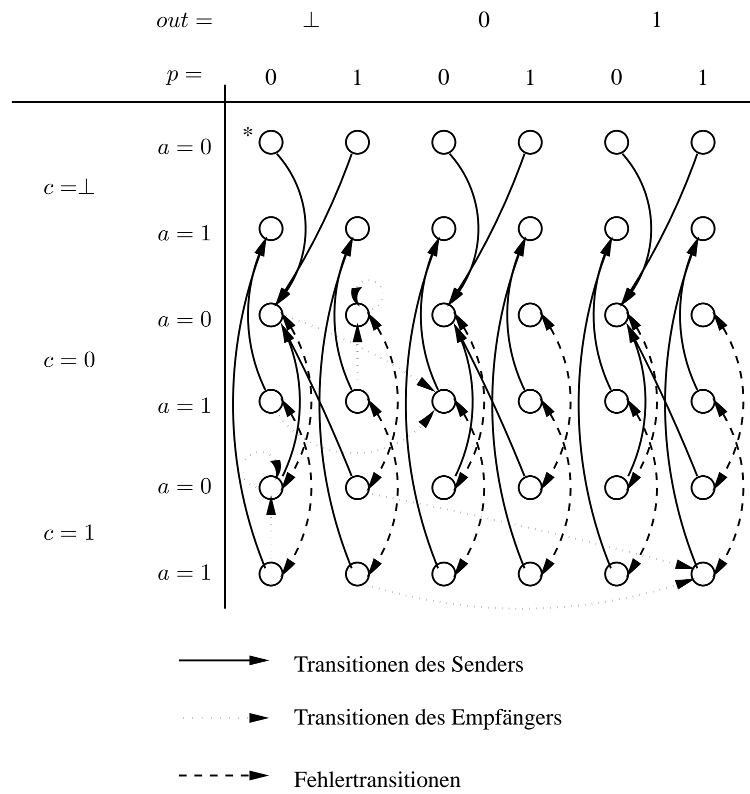


Abbildung 3.13: Zustandsraum des ARQ-Programms.

3.9 Diskussion

Zeitredundanz vs. Speicherredundanz. Sicherheit und Lebendigkeit werden oft als fundamental unterschiedliche, aber gleichwertige Konzepte aufgefaßt. Bei der Betrachtung der Ergebnisse aus den Abschnitten 3.6 und 3.7 ist jedoch eine Art von Asymmetrie zu beobachten, die sich auch in der Arora/Kulkarni-Theorie wiederfindet. Zeitredundanz alleine reicht nicht aus, um Lebendigkeit zu erreichen. Man benötigt zusätzlich Speicherredundanz. Dies ist jedoch eine natürliche Folgerung aus der Definition dieser beiden Begriffe. Die Asymmetrie tritt auch in der Arora/Kulkarni-Theorie auf, denn ein Korrektor enthält immer auch einen Detektor.

3.35 Theorem Hat ein Programm Zeitredundanz, dann hat es auch Speicherredundanz.

Beweis

ANNAHME: $\Sigma = (C, I, T, A)$ ist ein Programm mit Zeitredundanz.

ZEIGE: Σ hat Speicherredundanz.

1 $\langle 1 \rangle 1.$ Σ hat nicht-erreichbare Transitionen.
BEWEIS: Folgt aus der Voraussetzung und der Definition von Zeitredundanz. □

2 $\langle 1 \rangle 2.$ Es existiert eine Transition $t = (s, s') \in T$, für die gilt: In allen Abläufen σ von Σ kommt t nicht vor.
BEWEIS: Folgt aus Schritt $\langle 1 \rangle 1$ und der Definition einer nicht-erreichbaren Transition. □

3 $\langle 1 \rangle 3.$ In allen Abläufen σ von Σ kommt der Zustand s nicht vor.
BEWEIS: Folgt aus Schritt $\langle 1 \rangle 2$. □

4 $\langle 1 \rangle 4.$ s ist ein nicht-erreichbarer Zustand.
BEWEIS: Folgt aus Schritt $\langle 1 \rangle 3$ und der Definition eines nicht-erreichbaren Zustandes. □

5 $\langle 1 \rangle 5.$ Q.E.D.
BEWEIS: Die Konklusion des Theorems folgt aus Schritt $\langle 1 \rangle 4$ und der Definition von Speicherredundanz. □

Keine Fehlertoleranz ohne Redundanz. Oft wird plakativ die Meinung vertreten: "Es gibt keine Fehlertoleranz ohne Redundanz." Redundanz wird dabei als "definitorischer Bestandteil" eines Fehlertoleranzmechanismus aufgefaßt. Die in diesem Kapitel gewonnenen Erkenntnisse erlauben nun sogar, diese Aussage unter den gegebenen Voraussetzungen formal nachzuweisen. Dies zeigt, daß die Redundanzdefinitionen dieser Arbeit auch bezüglich dieser Beobachtung (bzw. Forderung) mit der Realität nicht im Widerspruch stehen.

3.36 Theorem (keine Fehlertoleranz ohne Redundanz) Seien Σ_1 und Σ_2 Programme, F ein sicherheits- und lebendigkeitskonservatives Fehlermodell, *SPEC* eine Spezifikation, und Σ_2 die F-tolerante Version von Σ_1 für *SPEC*. Dann ist Σ_2 redundant.

Beweis

ANNAHME:
1. Σ_1 und Σ_2 sind Programme,
2. F ist ein sicherheits- und lebendigkeitskonservatives Fehlermodell,
3. *SPEC* ist eine Spezifikation,
4. Σ_2 ist die F-tolerante Version von Σ_1 für *SPEC*.

ZEIGE: Σ_2 hat Redundanz.

1 $\langle 1 \rangle 1$. *SPEC* ist der Schnitt aus einer fusionsabgeschlossenen Sicherheitsspezifikation *SSPEC* und einer Lebendigkeitsspezifikation *LSPEC*.
BEWEIS: Folgt aus dem Theorem von Alpern und Schneider [12]. □

2 $\langle 1 \rangle 2$. Entweder $SSPEC \neq false$ oder $LSPEC \neq false$.
BEWEIS: Da *SPEC* nach Voraussetzung verletzbar ist, existiert wenigstens ein Ablauf, der nicht in *SPEC* ist, d.h. $SPEC \neq false$. Der Rest folgt aus Schritt $\langle 1 \rangle 1$. □

3 $\langle 1 \rangle 3$. ANNAHME: $SSPEC \neq false$
ZEIGE: Q.E.D.
BEWEIS: Die Konklusion folgt aus Theorem 3.28. □

4 $\langle 1 \rangle 4$. ANNAHME: $LSPEC \neq false$
ZEIGE: Q.E.D.
BEWEIS: Die Konklusion folgt aus Theorem 3.34. □

5 $\langle 1 \rangle 5$. Q.E.D.
BEWEIS: Die Konklusion folgt in jedem Fall, da Schritte $\langle 1 \rangle 3$ und $\langle 1 \rangle 4$ wegen der Feststellung von Schritt $\langle 1 \rangle 2$ alle Fälle abdecken. □

Redundanz als Folge der Fehlermöglichkeit. Wie Theorem 3.28 zeigt, ist Speicherredundanz ein zentrales Konzept beim Entwurf fehlertoleranter Systeme. Oft wird dieses Theorem mißverstanden, da bei genauerem Hinsehen nicht nur die fehlertolerante Version Σ_2 Speicherredundanz besitzt, sondern auch das fehler-intolerante Programm Σ_1. Der Einwand lautet: Wie kann man hier von "Einbringen von Redundanz" sprechen, wenn die Redundanz bereits vorher im System vorhanden ist?

Es gibt mindestens zwei Entgegnungen auf diesen Einwand. Die erste Entgegnung bezieht sich auf die konkrete Aussage des Theorems: Redundanz ist nicht

hinreichend, sondern nur *notwendig*. Dem Text des Theorems widerspricht es also nicht, wenn auch ein fehler-intolerantes Programm Speicherredundanz besitzt. Dennoch bleibt ein gewisses Unbehagen: Angenommen, man besitzt eine einzige Festplatte und möchte sich vor einem dauerhaften Datenverlust schützen, der durch einen *head crash* verursacht wird. Die Entwurfsmethodik, die der Arora/Kulkarni-Theorie (und auch Definition 3.17) zugrundeliegt, nimmt einfach an, es stünden plötzlich weitere Festplatten "wie aus dem nichts" zur Verfügung, mit deren Hilfe man Fehlertoleranz erreichen kann. Wie kann dies sein?

Die zweite Entgegnung holt etwas weiter aus und bemüht die Theorie der Verfeinerung (*refinement*). Diese Theorie besagt: Spezifikationen und Implementierungen können als logische Formeln im gleichen Formalismus ausgedrückt werden. Implementierungen sind sozusagen "etwas konkretere" Spezifikationen. Die Vor- und Nachbedingungen des *Sortierproblems* lauten zum Beispiel:

- Vorbedingung: Gegeben ein Feld a von k natürlichen Zahlen.
- Nachbedingung: Die Zahlen in a sind aufsteigend sortiert.

In einem mathematischen Formalismus ausgedrückt, wäre dies die Spezifikation eines C-Programms, welches beispielsweise *Quicksort* implementiert. Das C-Programm kann auf der anderen Seite wieder als Spezifikation angesehen werden, und zwar als Spezifikation des Maschinenprogramms auf einer bestimmten Rechnerarchitektur.

Im allgemeinen kann also die Programmentwicklung als eine Folge $S_1, \ldots, S_n$ von Spezifikationen angesehen werden, wobei S_1 die ursprüngliche Problemstellung ist und S_n der ausführbare Programmcode. Für jedes Paar (S_i, S_{i+1}) muß gelten: S_{i+1} implementiert S_i. Dies wird oft als *Verfeinerungsrelation* bezeichnet (*refinement*) [1]. Bei jedem Verfeinerungsschritt dürfen neue Variablen eingeführt werden, die für die jeweils höhere Ebene unsichtbar sind [113]. Bei den in Σ_1 vorhandenen zusätzlichen Zuständen handelt es sich um derartige Zustände.

An dieser Stelle ergibt sich eine interessante Einsicht: Die redundanten Zustände sind tatsächlich eine direkte Folge aus der Annahme, daß in Σ_1 Fehler auftreten können und diese in einer Erweiterung Σ_2 toleriert werden können. Ohne redundante Zustände gäbe es nur nicht-tolerierbare Fehlermodelle.

Relation zu Kodierungstheorie. Aus der Kodierungstheorie sind viele Verfahren bekannt, bei denen man durch die Hinzufügung von Bits zu Nachrichten Fehlererkennung und Fehlerkorrektur betreiben kann [153]. Die vielleicht einfachste Form ist das sogenannte *parity bit*: An einen Bitstrom wird mittels der Booleschen Operation XOR eine 1 angefügt, wenn die Anzahl der 1-Bits ungerade ist, andernfalls eine 0. Hierdurch können 1-Bit-Fehler bei der Datenübertra-

gung erkannt werden. Eine Verallgemeinerung sind die sogenannten *Hamming-Kodes* [153]. Speicherredundanz ist eine Verallgemeinerung dieser Form von Redundanz. Die Ergebnisse dieser Arbeit zeigen, daß die Wirkungsweise der Verfahren aus der Kodierungstheorie dieselbe ist wie die Funktionsweise praktisch aller Fehlertoleranzverfahren. Eine Begrifflichkeit der Zeitredundanz ist jedoch auch im Rahmen der Kodierungstheorie neu.

Auch im Bereich der Kryptographie wird eine Form von Speicherredundanz verwendet. Mit Methoden des *secret sharing* wird erreicht, daß ein Geheimnis auf n Personen in einer Art und Weise aufgeteilt werden kann, daß jeweils m Personen ($m < n$) genügen, um das Geheimnis wiederherzustellen [138]. Ein Verfahren, bei dem explizit auf den Zusammenhang zwischen Geheimnisschutz (*security*) und Fehlertoleranz eingegangen wird, wurde von Rabin [152] vorgestellt. Wir vermuten, daß zur Modellierung der Schutzaspekte in Bezug auf *security* die Definition von Speicherredundanz nicht ausreicht, denn kryptographische Verfahren können dies auch ganz ohne Speicherredundanz (z.B. durch Permutationschiffren) [138]) erreichen.

Automatisierung der Transformation. Kulkarni und Arora [108] haben kürzlich eine Methode vorgestellt, mit der man Detektoren und Korrektoren im Rahmen eine mechanischen Transformation automatisch einem gegebenen fehler-intoleranten Programm hinzufügen kann. Die vorgestellten Prozeduren basieren auf einer Exploration des Zustandsraums des fehler-intoleranten Programms. Dabei wird davon ausgegangen, daß ausreichende Speicherredundanz bereits im Programm vorhanden ist (vgl. Anmerkung oben). Ist dies nicht der Fall, bricht die Transformation mit der Meldung "fehlertolerantes Programm existiert nicht" ab. Außerdem werden nur eingeschränkte Lebendigkeitsspezifikationen der Form $\Diamond SPEC$ und eine restriktive Definition eines Fehlermodells betrachtet. So wird erstens angenommen, daß Fehler nur Transitionen hinzufügen und keine Auswirkungen auf die Lebendigkeitsannahme haben. Zweitens wird postuliert, daß Fehler nach endlicher Zeit aufhören. Der beschriebene Transformationsprozeß ist jedoch identisch mit dem in dieser Arbeit verwendeten Verständnis.

3.10 Zusammenfassung

Die Vielfalt an verschiedenen Fehlertoleranzmethoden ist auch für Experten heute kaum mehr zu überblicken. Beim Lesen von neuen Veröffentlichungen beschleicht einen jedoch oft das Gefühl, daß viele solche Verfahren einfache Neuanwendungen ein und desselben Inventars an Grundmechanismen der Fehlertoleranz sind. Die Fehlertoleranztheorie von Arora und Kulkarni [14] unterstützt diese Vermutung: Sie zeigt, daß unter bestimmten Annahmen nahezu jeder Fehlertoleranzmechanis-

mus als die Komposition eines fehler-intoleranten Programmes mit zusätzlichen Fehlertoleranzkomponenten beschrieben werden kann.

In diesem Kapitel haben wir die Arora/Kulkarni-Theorie näher vorgestellt und die genauen Annahmen untersucht, unter denen die Resultate der Theorie gelten. Wir haben die Theorie durch die Einführung einer Begrifflichkeit der *Redundanz* erweitert. Im Kontext der Fehlertoleranz ist dies nach bestem Wissen des Autors die erste vollständig formale Definition dieses Konzeptes. Es wurden zwei grundlegende Arten von Redundanz identifiziert (Speicherredundanz und Zeitredundanz) und ihr Verhältnis zu Sicherheits- und Lebendigkeitsspezifikationen nachgewiesen (vergleiche Tabelle 3.2). Außerdem wurden Aussagen bezüglich der Natur von tolerierbaren und nicht-tolerierbaren Fehlern im verwendeten Systemmodell gemacht.

fehlertolerant bezüglich	notwendig	Anforderung an F
Sicherheit	Speicherredundanz	sicherheitskonservativ
Lebendigkeit	Speicherredundanz und Zeitredundanz	sicherheitskonservativ und lebendigkeitskonservativ

Tabelle 3.2: Zusammenfassung der Ergebnisse.

Ausblick. Es liegt in der Natur der Sache, daß die Resultate dieses Kapitels nur unter den angegebenen Annahmen des zugrundeliegenden Systemmodells gelten. Wie in Kapitel 2 bereits erläutert, gibt es kein "richtiges" Systemmodell für fehlertolerante Systeme. Die Aussagekraft der Resultate im Hinblick auf Systemmodelle, die andere Abstraktionen verwenden, muß darum im Einzelfall jeweils neu überprüft werden. Steht beispielsweise ein in Hardware implementierter *voter* zur Verfügung, sind unter Umständen Fehlertoleranzmethoden wie TMR auch ohne einen Detektor im Sinne von Arora und Kulkarni möglich. Programmiert man den *voter* aus, muß es jedoch — so ein Ergebnis dieses Kapitels — einen Fehlererkennungszustand (d.h. Speicherredundanz) geben.

Im Einzelfall kann ein anderes Systemmodell auch zu neuen Redundanzdefinitionen führen. In nachrichtenbasierten Systemen gibt es beispielsweise den Begriff der *Nachrichtenredundanz*, die in verschiedenen Ausprägungen in *broadcast*-Algorithmen zum Einsatz kommt (*flooding* im Gegensatz zu "normaler" Implementierung).

Im Zusammenhang mit modernen und "sicheren" Systemarchitekturen werden zunehmend Begriffe wie "intrusionstolerant" verwendet [128]. Dies soll andeuten, daß man Sicherheit (im Sinne von *security*) mit denselben Mitteln erreichen will wie Fehlertoleranz. In diesem Kapitel haben wir uns auf Eigenschaften

im Sinne von Sicherheit und Lebendigkeit eingeschränkt. Gerade quantitative Eigenschaften (wie Echtzeit-Anforderungen [2]) oder Vertraulichkeitsaspekte wie die Abwesenheit von Informationsfluß [131, 137] sind jedoch nur schwer oder gar nicht als Menge von Abläufen formalisierbar. Die Hoffnung, man könne die Methoden und Resultate einfach auf andere Domänen übertragen, ist darum oft nicht gerechtfertigt.

Kapitel 4

Fehlertolerantes Beobachten

4.1 Einführung

Rückblick und Motivation. Die Fehlertoleranztheorie aus Kapitel 3 bietet Einsichten, die etablierte Resultate auf dem Gebiet der Fehlertoleranz erklären hilft und sich erfolgreich bei der Entwicklung neuer Fehlertoleranzmethoden anwenden lassen. Eine solche Einsicht ist: Man kann Fehler nur erkennen, wenn ein Fehlereffekt Teil des Systemzustandes geworden sind. Fehler, die von normalen Programmaktionen nicht zu unterscheiden sind, kann man nicht verläßlich erkennen. Dies ist beispielsweise die intuitive Grundlage für das in Kapitel 2 besprochene FLP-Resultat.

Eine zweite Einsicht ist: Jedes Fehlertoleranzverfahren besteht aus einem Fehlererkennungs- und einem Fehlerbehebungsmodul. Fehlererkennung und Fehlerbehebung sind demnach zwei unterschiedliche Aufgaben, die unabhängig voneinander betrachtet und untersucht werden können. Verbunden mit der ersten Einsicht bedeutet Fehlererkennung, daß man wissen möchte, ob sich das System in einem gewissen unerwünschten Zustand befindet. Unter diesem Blickwinkel ist Fehlererkennung also eine Art Beobachtungsproblem: Beobachte das System und melde, wenn ein bestimmter (globaler) Zustand eingetreten ist bzw. wenn ein globales Prädikat auf Systemzuständen gilt. In diesem Kapitel geht es um grundlegende Techniken der Beobachtung in verteilten Systemen, in denen Fehler auftreten können.

Unsicherheit bei der Beobachtung. Die Arbeit dieses Kapitels basiert auf den Annahmen des asynchronen Systemmodells (vergleiche Kapitel 2). In diesem Systemmodell ist das Problem der Beobachtung bereits intensiv untersucht worden, aber natürlich unter der Annahme, daß keine Fehler auftreten. Selbst unter dieser Annahme hat sich herausgestellt, daß das Beobachtungsproblem schwieri-

ger ist, als es anfangs erscheint.

Es gibt viele verschiedene Arten von Prädikaten, und man kann mit Recht annehmen, daß manche einfacher zu entdecken sind als andere. Im allgemeinen Fall läßt sich die Frage nach der Gültigkeit eines Prädikates schon bei ganz einfachen Beispielen nicht mehr eindeutig global beantworten. Vereinfachend gesagt liegt das daran, daß "annähernd gleichzeitige" Ereignisse auf verschiedenen Prozessen von unterschiedlichen Beobachtern in einer unterschiedlichen Reihenfolge gesehen werden können. Antworten auf die Frage nach der Gültigkeit eines Prädikates hängen also oft vom Beobachter ab. Wenn man aber gerne unabhängig vom Beobachter entscheiden möchte, ob ein Prädikat gilt oder nicht, dann bedient man sich sogenannter *Beobachtungsmodalitäten*: Man fragt beispielsweise nicht mehr "Gilt das Prädikat?" sondern: "Könnte es einen Beobachter geben, der dieses Prädikat erkennt?" Wenn allerdings *crash*-Fehler auftreten dürfen, stellen sich viele knifflige Fragen. Zum Beispiel: Wenn ein Prädikat φ auf einem Prozeß gilt und dieser Prozeß abstürzt, gilt φ dann weiterhin? Angesichts der Unsicherheiten bei der *crash*-Fehlererkennung machen die Beobachtungsmodalitäten für fehlerfreie Systeme keinen Sinn mehr. Die Leitfrage dieses Kapitels lautet also: Welche sinnvollen alternativen Beobachtungsmodalitäten gibt es in asynchronen verteilten Systemen, in denen *crash*-Fehler auftreten können?

Ausblick auf dieses Kapitel. Die Beobachtungsmodalitäten, die für fehlerfreie verteilte Systeme bereits entwickelt wurden, werden eingangs kurz im Abschnitt 4.2 wiederholt. In Abschnitt 4.3 werden einige der kniffligen Fragen diskutiert, die mit der Entdeckbarkeit von Prädikaten in fehlerbehafteten Systemen zusammenhängen. Wenn *crash*-Fehler im System auftreten können, werden nämlich die ursprünglich beobachterinvarianten Modalitäten wieder beobachterabhängig und verlieren somit ihren wesentlichen Vorteil. In Abschnitt 4.4 werden zwei neue Modalitäten vorgeschlagen, die auf die Eigenheiten des asynchronen Systemmodells und der *crash*-Fehlerannahme zugeschnitten sind. Diese neuen Modalitäten wären nutzlos, wenn man sie in derartigen Systemen nicht entdecken könnte. In den Abschnitten 4.5 und 4.6 wird gezeigt, daß dem nicht so ist: Abschnitt 4.5 stellt einfache Entdeckungsalgorithmen für die neuen Modalitäten vor, die allerdings auf der Annahme basieren, daß *beobachtende* Prozesse (im Gegensatz zu den *beobachteten*) nicht abstürzen dürfen. In Abschnitt 4.6 wird diese Restriktion fallengelassen und es werden Entdeckungsalgorithmen vorgestellt, die auch noch funktionieren, wenn auch die Beobachter abstürzen dürfen. Abschnitt 4.7 faßt die Ergebnisse dieses Kapitels nochmals ausführlich zusammen.

Das in diesem Kapitel verwendete Systemmodell ist das Fehlerdetektormodell, welches bereits ausführlich in Kapitel 2 besprochen worden ist. Kenntnis der dort behandelten Themen und Terminologie wird im folgenden vorausgesetzt.

4.2 Beobachten in fehlerfreien asynchronen Systemen

Applikationsprozesse und Monitorprozesse. Ein Beobachtungsbegriff basiert in der Regel auf einer Unterscheidung zwischen sogenannten *Applikationsprozessen* (*application processes*) und *Monitorprozessen* (*monitor processes*). Die Applikationsprozesse sind ganz normale Prozesse, welche die zu beobachtende verteilte Berechnung (die sogenannte *Basisberechnung*) ausführen. Dazu versenden sie untereinander ganz gewöhnliche Nachrichten (sogenannte *Applikationsnachrichten*, *application messages*). Die Aufgabe der Monitorprozesse ist, die Berechnung zu beobachten. Monitorprozesse werden deshalb auch oft als *Beobachter* oder *Beobachterprozesse* (*observers*, *observation processes*) bezeichnet. Dazu wird ein anderer Typ von Nachrichten, sogenannte *Kontrollnachrichten* (*control messages*) verwendet: Wann immer ein nennenswertes Ereignis auf einem Applikationsprozeß passiert, schickt der Applikationsprozeß eine Kontrollnachricht an alle Beobachter. Das System muß so konzipiert sein, daß die Basisberechnung nicht gestört wird, auch wenn sie unter Beobachtung steht. Im folgenden wird von n Applikationsprozessen $p_1, \ldots, p_n$ und m Beobachtungsprozessen $b_1, \ldots, b_m$ ausgegangen.

Das Beobachtungsproblem. Ausgangspunkt für die Beobachtung ist ein Prädikat auf Systemzuständen, welches man im verteilten System entdecken will. Formal ist ein Prädikat eine Abbildung von der Menge der globalen Systemzustände in die Menge $\{true, false\}$. Die Betrachtung beschränkt sich auf Prädikate, die sich lediglich auf den Zustand von Prozessen beziehen und nicht auch noch auf den Zustand von Kommunikationskanälen. (Wenn man von fehlerfreien Kanälen ausgeht, kann man den Zustand der Kanäle durch Zähler bzw. durch Zwischenspeichern der versendeten und empfangenen Nachrichten innerhalb der Prozesse mitmodellieren.) Beispiele für derartige Prädikate sind:

- Jeder Prozeß befindet sich im Zustand “terminiert”.
- Prozeß p_1 wartet auf eine Nachricht von p_2 und p_2 wartet auf eine Nachricht von p_1 (d.h. im System herrscht eine Verklemmungssituation).
- Die Gesamtlast des Systems übersteigt einen bestimmten Wert.
- Es gibt kein *token* mehr im System.
- Obwohl Prozeß p_1 das *token* anforderte, hat er innerhalb der letzten zehn Ausführungsschritte das *token* nicht erhalten.

Wir betrachten nun die Ausführung eines verteilten Systems und möchten gerne wissen, ob das Prädikat während dieser Ausführung gilt oder nicht. Das *Beobachtungsproblem* (*predicate detection problem* [36]) wird dazu wie folgt beschrieben: Finde einen verteilten Algorithmus, der die folgenden beiden Eigenschaften erfüllt:

- (Sicherheit) Es ist nie der Fall, daß der Algorithmus "Prädikat φ gilt" meldet ohne daß φ auch tatsächlich gilt.
- (Lebendigkeit) Falls φ im Verlauf der Ausführung gilt, meldet der Algorithmus dies nach endlicher Zeit.

Die Annahme hinter dieser Spezifikation ist, daß die zugrundeliegende Basisberechnung nicht terminiert. Diese Form der Prädikatsentdeckung wird oft als *online* bezeichnet. Wenn die Basisberechnung terminiert, muß man zusätzlich fordern, daß der Algorithmus nach Beendigung der Basisberechnung "Prädikat φ gilt nicht" meldet, falls im Laufe der Ausführung φ nie galt (dies wird dann als *offline predicate detection* bezeichnet). Dieses Kapitel beschränkt sich auf *online*-Prädikatserkennung, da es sich dabei um ein allgemeineres Problem handelt (d.h. wenn man *online*-Prädikatserkennung lösen kann, dann kann man es auch *offline* tun).

In diesem Kapitel liegt das Interesse auf *beobachterunabhängigen* Lösungen des Beobachtungsproblems, d.h. es sollte nie der Fall sein, daß zwei Beobachter, welche den gleichen Algorithmus ausführen, zu unterschiedlichen Entscheidungen über die Gültigkeit des Prädikates kommen.

Ereignisse, Kausalität und konsistente globale Zustände. Die Ausführung der Basisberechnung erzeugt auf jedem Applikationsprozeß eine Folge $e_1, e_2, \ldots$ von lokalen Zustandsübergängen. Ein Zustandsübergang wird oft auch als *Ereignis* (*event*) bezeichnet. Um die Ereignisse auf verschiedenen Prozessen zu unterscheiden, werden hochgestellte Ziffern verwendet, d.h. e_i^k bezeichnet das i-te Ereignis auf Prozeß p_k. Über die zeitliche Relation dieser Ereignisse kann man im zugrundeliegenden asynchronen Systemmodell kaum Aussagen treffen. Man kann allerdings eine *Kausalitätsrelation* definieren, die auf einer potentiellen Ursache-Wirkungs-Beziehung basiert [112]. Die Kausalitätsrelation ist eine partielle Ordnung $\rightarrow$ auf der Menge der Ereignisse. Es ist die kleinste Relation, die folgende drei Bedingungen erfüllt:

1. Für zwei Ereignisse e_i^k und e_j^k auf demselben Prozeß gilt $e_i^k \rightarrow e_j^k$ falls $i+1=j$.
2. Für zwei Ereignisse e_i^k und e_j^l auf (möglicherweise) unterschiedlichen Prozessen gilt $e_i^k \rightarrow e_j^l$ falls e_j^k das Senden einer Nachricht m ausdrückt und e_j^l

den Empfang von m darstellt.

3. Die Relation $\rightarrow$ ist transitiv.

Eine verteilte Berechnung läßt sich durch ein *Raum/Zeit-Diagramm* (*space/time diagram*) darstellen [165] (vergleiche Abbildung 4.1). Darin sind die einzelnen Prozesse durch horizontale Zeitachsen dargestellt (die Zeit vergeht "von links nach rechts") und Ereignisse durch Punkte auf den Achsen. Zusammengehörige Sende- und Empfangsereignisse werden durch Pfeile verbunden, die den Weg der Kommunikation anzeigen. (Beispielsweise kommt die Nachricht, die in Ereignis e_1^1 losgeschickt wurde, in Ereignis e_1^3 an, und Ereignis e_2^3 ohne Pfeil steht für ein internes Ereignis.)

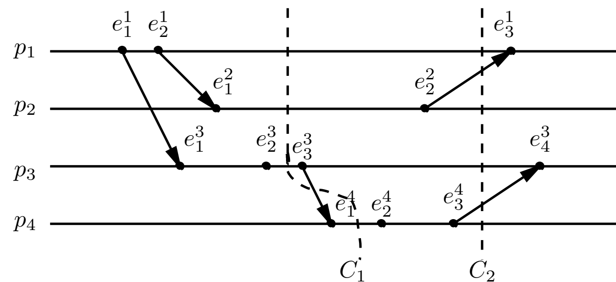


Abbildung 4.1: Das Raum/Zeit-Diagramm einer verteilten Berechnung mit einem inkonsistenten Schnitt C_1 und einem konsistenten Schnitt C_2.

Zwischen zwei Ereignissen verharrt ein Prozeß in einem *lokalen Zustand* (*local state*). Ein *globaler Zustand* (*global state*) ist ein n-Tupel $S = (s_1, s_2, \ldots, s_n)$ von lokalen Zuständen, wobei s_i den lokalen Zustand von p_i bezeichnet. Ein solcher globaler Zustand entspricht immer einem *Schnitt* (*cut*) durch das Raum/Zeit-Diagramm der Berechnung [136, 165]. Ein Schnitt ist eine gedachte Linie durch das Raum/Zeit-Diagramm, die direkt hinter denjenigen Ereignissen e_k^i verläuft, die jeweils zum Zustand s_i auf jedem Prozeß geführt haben (vergleiche Schnitte C_1 und C_2 in Abbildung 4.1). Ein Schnitt beschreibt außerdem wieder eine Ereignismenge, d.h. die Menge aller Ereignisse, die auf jedem Prozeß zum Zustand s_i geführt haben.

Basierend auf der Kausalitätsrelation kann man nun globale Zustände in konsistente und inkonsistente Zustände einteilen. Ein Zustand ist *konsistent* genau dann, wenn die Ereignismenge des dazugehörigen Schnittes links-abgeschlossen bezüglich $\rightarrow$ ist. Eine Menge M ist links-abgeschlossen bezüglich $\rightarrow$, wenn folgendes gilt: Falls $e \in M$ und $e' \rightarrow e$, dann ist $e' \in M$. Entsprechend ist der durch den Schnitt C_1 definierte Zustand in Abbildung 4.1 inkonsistent, da e_1^4 Teil des Schnittes ist aber e_2^3 nicht. In konsistenten Zuständen sind demnach alle empfangenen Nachrichten auch gesendet worden. (Der durch Schnitt C_2 in

Abbildung 4.1 definierte Zustand ist ein Beispiel für einen konsistenten Zustand.) Es macht also Sinn, sich bei Beobachtungen auf die Darstellung von konsistenten Zuständen zu beschränken.

Beobachtungen, Zustandsverände und Beobachterabhängigkeit. Im Verlauf einer Berechnung empfängt ein Beobachterprozeß also eine Folge von Kontrollnachrichten von jedem Applikationsprozeß und versucht, aus den darin enthaltenen Informationen die Basisberechnung zu rekonstruieren. Ein Beobachter konstruiert also eine Folge $\Sigma = S_1, S_2, \ldots$ von globalen Zuständen. Man nennt eine solche Folge auch *Beobachtung* (*observation*). Eine Beobachtung heißt *konsistent* (manchmal auch *kausal-konsistent*), wenn jeder Zustand S_i ein konsistenter Zustand ist und wenn jeder Zustand S_{i+1} durch genau ein Ereignis aus dem Zustand S_i hervorgegangen ist. In diesem Fall sagt man, daß S_i ein *Vorgänger* von S_{i+1} ist.

Die Implementierung von *kausal-konsistenten Beobachtern* (d.h. Beobachtern, die ausschließlich kausal-konsistente Beobachtungen generieren) ist bereits recht gut erforscht. Die dafür eingesetzten Mechanismen beruhen immer auf sogenannten *Vektoruhren* (*vector clocks*) [67, 90, 91, 136, 165]. Vektoruhren erzeugen Zeitstempel, die die partielle Kausalitätsordnung in isomorpher Weise abbilden. Derartige Zeitstempel können beispielsweise zur Implementierung eines kausalen *broadcast* verwendet werden [91].

Man kann zeigen, daß die Menge aller konsistenten Zustände zusammen mit der oben beschriebenen Vorgänger-Relation einen *Verband* (*lattice*) bilden [21, 136, 165]. Zur Erinnerung: Zu einem Verband gehören eine Menge M, eine partielle Ordnung auf M und die zweistelligen Operationen *sup* und *inf* auf M. Diese Struktur ist genau dann ein Verband, wenn für alle Paare a, b aus M sowohl $sup(a, b)$ als auch $inf(a, b)$ existieren [30]. Im Falle der Menge der konsistenten Zustände einer verteilten Berechnung entsprechen die Operationen *inf* und *sup* zum einen dem Schnitt und zum anderen der Vereinigung der den Zuständen entsprechenden Ereignismengen. Der initiale Zustand des Systems bildet das *top*-Element des Verbandes (der meistens jedoch "unten" gezeichnet wird). Bei einer nichtterminierenden Berechnung hat man es im allgemeinen mit unendlich vielen globalen Zuständen zu tun, d.h. mit einem unendlichen Verband. Man kann jedoch diesen unendlichen Verband zu jedem Zeitpunkt jeweils durch einen endlichen Verband approximieren, in den jeweils am "unteren" Rand (d.h. unterhalb des *bottom*-Elementes) neue Elemente eingefügt werden, wenn die Berechnung fortschreitet. Endliche Verbände habe die erfreuliche Eigenschaft, daß man sie graphisch darstellen kann. Bei der graphischen Darstellung kann man sich zunutze machen, daß der Verband der konsistenten Zustände ein Unterverband vom Verband *aller* möglichen Zustände einer Berechnung ist (ein n-dimensionaler Hyperwürfel). Das bedeutet aber auch, daß sich Berechnungen mit mehr als drei

Prozessen nur noch schwer auf Papier darstellen lassen.

Jede konsistente Beobachtung entspricht einem Pfad vom *top*- zum *bottom*-Element im Verband der konsistenten Zustände. Es kann nun sein, daß ein zu entdeckendes Prädikat φ nur an ganz wenigen globalen Zuständen wahr ist. Läuft eine Beobachtung durch einen solchen Zustand, wird der entsprechende Beobachter φ entdecken, andernfalls nicht. Die Frage, ob ein Prädikat φ im Verlauf einer Beobachtung gilt, ist also generell beobachter*abhängig*. Es macht also im allgemeinen Fall keinen Sinn zu fragen, ob ein Prädikat im Laufe einer Berechnung gilt oder nicht.

Die Beobachtungsmodalitäten *possibly* und *definitely*. Basierend auf dem Verband der konsistenten Zustände ist es möglich, beobachterunabhängige Gültigkeitsformen zu definieren. Man bezieht die Frage nach der Gültigkeit nicht mehr auf eine einzelne Beobachtung, sondern auf die Gesamtheit aller möglichen konsistenten Beobachtungen. Das dabei verwendete Konstrukt wird manchmal *Prädikatstransformator* (*predicate transformer*) [37, 157] oder *Beobachtungsmodalität* (*observation modality*) [21, 165] genannt.

Ausgehend von einem globalen Prädikat φ haben sich die folgenden beiden Modalitäten als nützlich erwiesen [41, 133]:

- *possibly*(φ) gilt genau dann, wenn eine konsistente Beobachtung $\Sigma = S_1, S_2, \ldots$ existiert, so daß für eine dieser globalen Zustände S_i das Prädikat φ gilt.
- *definitely*(φ) gilt genau dann, wenn in allen konsistenten Beobachtungen $\Sigma = S_1, S_2, \ldots$ ein Zustand S_i existiert, in dem φ gilt.

Die Modalität *possibly*(φ) wird also genau dann entdeckt, wenn es einen Beobachter gibt, der φ entdeckt haben könnte. Auf der anderen Seite wird *definitely*(φ) genau dann entdeckt, wenn alle Beobachter φ entdeckt hätten.

Die Modalitäten *possibly* und *definitely* haben eine gewisse Affinität zu bestimmten Formen von Sicherheits- und Lebendigkeitseigenschaften. Soll eine Berechnung beispielsweise die Eigenschaft $\Box\varphi$ besitzen, so sollte *possibly*$(\neg\varphi)$ nie entdeckt werden können. Falls es doch entdeckt würde, besitzt die Berechnung nicht diese Eigenschaft. Sollte hingegen eine Berechnung die Eigenschaft $\Diamond\varphi$ erfüllen, so ist es hinreichend *definitely*(φ) zu erkennen, um diese Eigenschaft für die konkrete Berechnung zu verifizieren.

Entdeckungsalgorithmen für *possibly* und *definitely*. In der Literatur existieren eine Reihe von Entdeckungsalgorithmen für *possibly* und *definitely* [21, 39, 41, 99, 165, 168, 169], die mit dem zu entdeckenden Prädikat φ initialisiert

werden müssen. Die laufende Eingabe für diese Algorithmen besteht aus einer Folge von Ereignissen, die gemäß $\rightarrow$ vorsortiert sein müssen. Die Algorithmen lösen das Beobachtungsproblem für das (transformierte) Prädikat $possibly(\varphi)$ beziehungsweise $definitely(\varphi)$. Im Kern basieren die Algorithmen auf einer Konstruktion des oben beschriebenen Verbandes konsistenter Zustände. Entsprechend haben diese Algorithmen entweder eine exponentielle Laufzeit [99] oder exponentiellen Speicherverbrauch [41]. Dies ist nicht verwunderlich, denn Chase und Garg haben in einem lesenswerten Überblicksartikel [39] nachgewiesen, daß das Problem der Entdeckung allgemeiner Prädikate in asynchronen Systemen NP-vollständig ist.

4.3 Beobachten in fehlerbehafteten asynchronen Systemen

Die im vergangenen Abschnitt vorgestellten Modalitäten *possibly* und *definitely* sind für die Verwendung in fehlerfreien verteilten Systeme definiert worden. In diesem Abschnitt geht es darum, welche Probleme sich bei der Übertragung dieser Prädikatstransformatoren in Systeme mit *crash*-Fehlern ergeben und wie sie gelöst werden können.

4.3.1 Bisherige Arbeiten

Es gibt bereits mehrere Arbeiten, die sich mit der Erkennung von Prädikaten in fehlerbehafteten verteilten Systemen beschäftigen [70, 177, 184]. Während Li und McMillin [184] sowie Venkatesan [177] perfekte Fehlerdetektoren benutzen, basieren die Algorithmen von Garg und Mitchell [70] bisher als einzige auf dem Konzept der unzuverlässigen Fehlerdetektoren. Garg und Mitchell schränken aber die Menge der entdeckbaren Prädikate ein auf sogenannte *Mengenredunktionsprädikate* (*set decreasing predicates*). Ein Prädikat ϕ heißt *mengenreduzierend*, wenn folgendes gilt: Wenn ϕ für eine Menge von Prozessen gilt, dann gilt ϕ auch für jede Teilmenge dieser Prozesse. Das Prädikat "kein *token* vorhanden" ist ein Beispiel für ein solches Prädikat. Wie im folgenden erläutert, umgehen Garg und Mitchell durch die Einschränkung der Prädikate gerade die interessanten und schwierigen Punkte, die bei der Erkennung allgemeiner Prädikate in fehlerbehafteten verteilten Systemen eine Rolle spielen.

4.3.2 Prädikate und ihre Bedeutung unter der *crash*-Fehlerannahme

Alle Prädikate, die eingangs von Abschnitt 4.2 genannt wurden, machen weiterhin Sinn in fehlerbehafteten Systemen. Wenn Fehler auftreten können, wird jedoch auch eine neue Klasse von Prädikaten interessant, für die in fehlerfreien Systemen keine Verwendungsgrund bestand, nämlich Prädikate, die sich auf die Auswirkung von Fehlern beziehen. Für die *crash*-Fehlerannahme wäre ein solches Prädikat beispielsweise: "Genau zwei Prozesse sind abgestürzt."

Um diese Form von Prädikaten ausdrücken zu können, muß der funktionale Zustand von Prozessen als Teil des lokalen Prozeßzustandes ausgedrückt werden können. Zu diesem Zweck wird angenommen, jeder Prozeß besitze einen Vektor $up[1..n]$ von Booleschen Werten. (Ein hochgestellter Index soll wieder die Zugehörigkeit des Vektors zu einem bestimmten Prozeß ausdrücken, d.h. up^k ist der Vektor von Prozeß p_k.) Der Zusammenhang zwischen dem Vektor und dem funktionalen Zustand von Prozessen ist der folgende: $up[i] = false$ genau dann, wenn Prozeß p_i abgestürzt ist. Angenommen, wir betrachten ein System mit drei Applikationsprozessen. Dann kann man mit Hilfe dieses Vektors das Prädikat γ, welches ausdrücken soll, daß genau zwei Prozesse abgestürzt sind, formalisieren als:

$$\begin{aligned} \gamma \equiv \; & (\neg up[1] \wedge \neg up[2] \wedge up[3]) \vee \\ & (\neg up[1] \wedge up[2] \wedge \neg up[3]) \vee \\ & (up[1] \wedge \neg up[2] \wedge \neg up[3]) \end{aligned}$$

Was passiert mit Prädikaten auf abstürzenden Prozessen? Wenn man davon ausgeht, daß Prozesse abstürzen können, dann muß man eine zusätzliche Frage diskutieren: Wenn ein Prozeß abstürzt, behalten dessen Variablen ihre Werte oder "verschwinden" sie irgendwie?

Der Hintergrund dieser Frage kann am folgenden Beispiel deutlich gemacht werden: Angenommen, Prozeß p_1 besitzt eine Variable x, die initial den Wert 1 hat. Das Prädikat, welches zu entdecken ist, lautet $\varphi \equiv x \neq 1$. Solange p_1 noch ordnungsgemäß läuft, so dürfte klar sein, daß φ nicht gilt, und darum sollte ein Entdeckungsalgorithmus dies auch nicht melden. Doch wie steht es um die Gültigkeit von φ, wenn p_1 plötzlich abstürzt? Soll der Algorithmus sich jetzt melden oder nicht?

Interessanterweise machen beide möglichen Alternativen Sinn. Falls beispielsweise der Wert $x = 1$ den Besitz eines *token* darstellt, dann würde $x \neq 1$ bedeuten, daß p_1 das *token* nicht mehr besitzt. Da p_1 abgestürzt ist und somit das *token* nicht mehr zurückgeben kann, wäre es durchaus sinnvoll, wenn der Algorithmus meldete "φ gilt". In diesem Fall kann die Entdeckung von φ die Generierung

eines neuen *token* auslösen.

Man könnte aber auch für die andere Variante argumentieren, in der der Algorithmus sich nicht meldet, wenn p_1 abstürzt. Beispielsweise könnte der Wert $x = 1$ den Entscheidungswert eines *consensus*-Algorithmus (vergleiche Kapitel 2) darstellen. Wenn der Algorithmus dann melden würde, daß $x \neq 1$ gilt und alle anderen Prozesse sich für den Wert 1 entschieden haben, dann könnte ein Beobachter fälschlicherweise zu dem Schluß kommen, die Sicherheit des *consensus*-Algorithmus sei verletzt.

Aus einer pragmatischen Sichtweise gibt es zwei Alternativen:

A1 Bei einem Absturz eines Prozesses p_i behalten seine Variablen einfach ihre Werte bei. Nur der Wert von $up[i]$ ändert sich von *true* nach *false*.

A2 Bei einem Absturz eines Prozesses p_i nehmen die Variablen einen undefinierten Wert $\perp$ an.

Bei genauerer Überlegung stellen sich bei der Variante A2 eine Fülle weiterer Fragen. Beispielsweise: Muß der Wert $\perp$ für alle Variablen verschieden sein oder nicht? Wenn man annimmt, $\perp$ sei der gleiche Wert für alle Variablen, dann würde der Algorithmus unter Umständen Prädikate wie $x = y$ fälschlicherweise entdecken, weil x und y bei einem Absturz beide den Wert $\perp$ annehmen würden. Außerdem: Wenn man schon ein besonderes Symbol $\perp$ für die Variablenwerte abgestürzter Prozesse einführt, warum braucht man dann überhaupt noch eine Variable *up*, wo doch $up[i] = false$ äquivalent zu $x_i = \perp$ ist (hier ist x_i eine beliebige Variable auf p_i)?

Aus einer abstrakteren Perspektive zeigt sich, daß die Alternative A1 (Variablen behalten ihren Wert) vollkommen ausreichend ist und wir uns deshalb für sie entscheiden können. Um das zu verstehen, betrachten wir zunächst sogenannte *lokale Prädikate* (*local predicates*) [37]. Ein Prädikat φ heißt *lokal* zu Prozeß p_i falls in φ nur Variablen aus p_i vorkommen.

Gegeben sei also ein lokales Prädikat φ_i von Prozeß p_i. Wenn Variablen bei einem Absturz ihren Wert behalten, entdecken wir implizit nicht φ_i sondern $up[i] \wedge \varphi_i$: Damit φ_i wahr werden kann, muß ein Ereignis auf p_i passieren. Dazu muß p_i am Leben sein, d.h. $up[i] = true$ gilt. Wenn man aber ein gegenteiliges Verhalten erzeugen möchte (der Algorithmus meldet sich, wenn p_i abstürzt), dann muß man explizit schreiben $\neg up[i] \vee \varphi_i$. Die Ausdruckskraft der Alternative A1 ist für lokale Prädikate also nicht eingeschränkt.

Ähnliches gilt für Konjunktionen oder Disjunktionen lokaler Prädikate, also Prädikate der Form $\varphi_i \wedge \varphi_j$ oder $\varphi_i \vee \varphi_j$. Ein Beispiel wäre das Prädikat "mindestens ein *token* im System", welches man als

$$x_1 = 1 \vee x_2 = 1 \vee \ldots \vee x_n = 1$$

formalisieren könnte (wobei $x_i = 1$ wieder bedeutet, daß p_i das *token* besitzt). Man ersetzt einfach standardmäßig das lokale Prädikat durch die Konjunktion mit der entsprechenden *up*-Variable, also:

$$(up[1] \wedge x_1 = 1) \vee (up[2] \wedge x_2 = 1) \vee \ldots \vee (up[n] \wedge x_n = 1)$$

Wenn nichts anderes spezifiziert wurde, wird diese Version des Prädikates erkannt.

Im allgemeinen können aber auch Prädikate entdeckt werden, die nicht lokal sind, zum Beispiel $x_1 \neq x_2$. Hier folgen wir wieder dem Credo "Variablen behalten ihren Wert": Wenn man möchte, daß der Algorithmus dieses Prädikat beim Absturz von p_1 oder p_2 erkennt, muß man schreiben:

$$\neg up[1] \vee \neg up[2] \vee x_1 \neq x_2$$

(Durch Weglassen des ersten Terms kann man sogar noch genauer spezifizieren, daß der Algorithmus sich nur beim Absturz von p_2 melden soll.) Die erste Alternative ist also vollkommen ausreichend und sogar relativ flexibel.

4.3.3 Beobachterabhängigkeit von *possibly* und *definitely*

Wenn man wie oben beschrieben einen *up*-Vektor auf jedem Prozeß annimmt, der Auskunft über den funktionalen Zustand entfernter Prozesse gibt, dann muß man auch einen Mechanismus haben, der die Werte von *up* immer auf dem neusten Stand hält. Die Funktionalität, die hier benötigt wird, ist die eines aus Kapitel 2 bekannten perfekten Fehlerdetektors. Zur Wiederholung: Ein perfekter Fehlerdetektor erfüllt die folgenden Eigenschaften:

- starke Genauigkeit (es ist nie der Fall, daß $up[i] = \mathit{false}$ ohne daß p_i auch tatsächlich abgestürzt ist), und
- starke Vollständigkeit (wenn p_i abstürzt, dann werden nach endlicher Zeit alle bis dahin laufenden Prozesse $up[i]$ auf *false* setzen).

Man kann nun (in Analogie zu einer verbreiteten Methode der Fehlermodellierung [75]) den Wert von $up[i]$ als Teil des Zustandes von Prozeß p_i betrachten. Die so vergrößerte Menge der Zustände wird *erweiterter Zustandsraum* (*extended state space*) genannt. Wenn der Fehlerdetektor also lokal meldet, daß p_i abgestürzt ist, dann kann man dies einfach als eine Kontrollnachricht interpretieren, die über den Zustandsübergang der *up*-Variable auf p_i von *true* nach *false* informiert. Mit Hilfe dieser Interpretation kann man die Prädikat-Detektions-Techniken aus dem fehlerfreien Fall auch in Umgebungen mit *crash*-Fehlern einsetzen: Jeder Beobachter baut wie im im fehlerfreien Fall einen Verband konsistenter Zustände und entdeckt darin *possibly* oder *definitely*.

Wie aus Kapitel 2 ebenfalls bekannt ist, kann man einen perfekten Fehlerdetektor aber in asynchronen Systemen nicht implementieren. Man kann entweder nur starke Genauigkeit oder starke Vollständigkeit erreichen, aber nicht beides gleichzeitig. Das heißt, Fehlerdetektoren werden unter Umständen falsche Verdächtigungen aussprechen oder einen tatsächlich abgestürzten Prozeß niemals verdächtigen. So können zwei Prozesse p_1 und p_2 mehr oder weniger unabhängig voneinander und davon, ob ein anderer Prozeß p_3 tatsächlich abgestürzt ist, zu unterschiedlichen Entscheidungen über den funktionalen Zustand von p_3 kommen. Bei der Frage, gilt das Prädikat $\varphi \equiv up[3] = \mathit{false}$, werden p_1 und p_2 also unterschiedliche Meinungen haben. Dies führt dazu, daß p_1 und p_2 auch bezüglich der Gültigkeit von $\mathit{possibly}(\varphi)$ oder $\mathit{definitely}(\varphi)$ verschiedener Meinung sein können. Die *crash*-Fehlerannahme erzeugt also neue Beobachterabhängigkeit, insbesondere haben die zuvor beobachterinvarianten Modalitäten *possibly* und *definitely* nicht mehr diese schöne Eigenschaft.

4.3.4 Fehlerdetektoren für die Prädikatserkennung

Zwar sind starke Genauigkeit und Vollständigkeit gleichzeitig nicht erreichbar, doch kann man in asynchronen Systemen etwas schwächere Fehlerdetektoreigenschaften realisieren, die trotzdem bei der Prädikatserkennung Sinn machen. Wie in Kapitel 2 angedeutet, kann man starke Vollständigkeit (kein Absturz wird übersehen) mittels einem zuverlässigen *broadcast* in asynchronen Systemen realisieren. Außerdem kann man erreichen, daß ein dauerhaft funktionierender Prozeß nicht dauerhaft von einem anderen Prozeß verdächtigt wird. Dies läßt sich dadurch erreichen, daß jeder Prozeß periodisch Lebenssignale an alle anderen Prozesse schickt und die Prozesse beim Eintreffen einer solchen Nachricht eine mögliche Verdächtigung des entfernten Prozesses widerrufen. Derartige Fehlerdetektoren werden *unendlich häufig genau* (*infinitely often accurate*) genannt [70].

Die orginale Definition eines Fehlerdetektors von Chandra und Toueg [35] macht keine Aussage über das kausale Verhältnis von Fehlerdetektormeldungen und Nachrichten, die von entfernten Prozessen eintreffen. Es kann also durchaus sein, daß der Fehlerdetektor einen Prozeß p_i verdächtigt, und anschließend noch Kontrollnachrichten von p_i eintreffen. Der strengen Trennung von Funktionalität und Implementierung folgend bedeutet dies *nicht*, daß der Fehlerdetektor die Verdächtigung widerrufen muß. Wenn man aber Prädikate in verteilten Systemen erkennen will, benötigt man einen Hinweis über die Position einer Verdächtigung innerhalb der kausalen Ordnung der eingehenden Kontrollnachrichten, da diese bei der Entdeckung von Prädikaten der Form “Prozeß p_i stürzt im Zustand x ab” von Bedeutung sind.

Die einfachste Möglichkeit, Fehlerdetektornachrichten mit den normalen Applikationsereignissen in Beziehung zu setzen, benutzt den Empfangszeitpunkt einer

Verdächtigung: Angenommen, ein Beobachter b_j hat von einem Prozeß p_i zuletzt Informationen über das Ereignis e_k^i erhalten. Dann wird eine anschließende Verdächtung "hinter" diesem Ereignis in die kausale Ordnung eingereiht. Wir sagen in diesem Fall, daß Beobachter b_j den Prozeß p_i *nach dem Ereignis* e_k^i *verdächtigt*. Natürlich ist es nicht möglich zu wissen, ob diese Verdächtigung der Wahrheit entspricht, d.h. ob noch eine Kontrollnachricht von Prozeß p_i unterwegs ist, die nach der Verdächtigung ankommt. Das einzige, was man tun kann, ist, die Verdächtigung nach dem Eintreffen einer normalen Kontrollnachricht von Prozeß p_i wieder zurückzunehmen. Wir sagen in diesem Fall, daß Beobachter b_j den Prozeß p_i *nach dem Ereignis* e_k^i *rehabilitiert*. Ein Beispiel für einen Ablauf, in dem ein Beobachter einen Prozeß (zunächst fälschlicherweise) verdächtigt und rehabilitiert, ist in Abbildung 4.2 dargestellt.

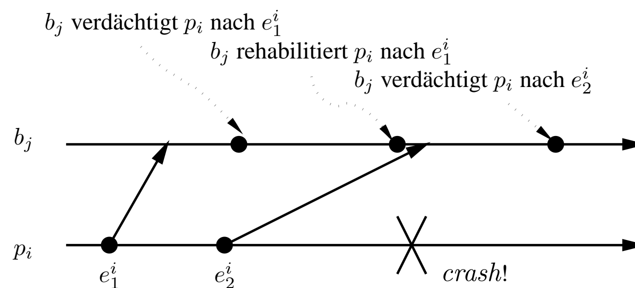


Abbildung 4.2: Verdächtigungen und Rehabilitierungen in einer verteilten Berechnung.

Eine Kette von wiederholten Verdächtigungen und Rehabilitierungen ist von ihrem Informationsgehalt her für die Prädikatserkennung uninteressant (in anderen Kontexten kann diese Information jedoch von Bedeutung sein, etwa bei Effizienzbetrachtungen). Wenn man einen Prozeß erst nach dem Eingang einer Kontrollnachricht rehabilitiert, hat man auch automatisch die Eigenschaft, daß es zwischen zwei Ereignissen auf einem Prozeß maximal eine Verdächtigung und eine Rehabilitierung gibt. Insgesamt ist es darum realistisch, die folgenden Eigenschaften eines Fehlerdetektors für die Prädikatserkennung zu fordern.

4.1 Definition (kausaler unendlich häufig genauer Fehlerdetektor) Ein *kausaler unendlich häufig genauer Fehlerdetektor* ist ein Fehlerdetektor, der die folgenden Eigenschaften erfüllt:

- (unendlich häufige Genauigkeit) Für alle korrekten Prozesse p_i gilt: p_i wird nie dauerhaft durch einen korrekten Prozeß p_j verdächtigt.

- (starke Vollständigkeit) Für alle korrekten Prozesse p_i gilt: Wenn ein Prozeß p_j abstürzt, dann wird p_j nach endlicher Zeit auf p_i verdächtigt.

- (maximal eine Verdächtigung) $\forall k$: $\forall i$: $\forall j$: b_j verdächtigt p_i nach e^i_k maximal ein Mal.
- (maximal eine Rehabilitierung) $\forall k$: $\forall i$: $\forall j$: b_j rehabilitiert p_i nach e^i_k maximal ein Mal und auch nur dann, wenn b_j zuvor p_i nach e^i_k verdächtigt hat.
- (garantierte Totzeit) Wenn p_j p_i nach e^i_k verdächtigte und ein Ereignis e^i_{k+1} existiert, dann rehabilitiert p_j p_i nach e^i_k bevor das Ereignis e^i_{k+1} an p_j zugestellt wird.

Die ersten beiden Eigenschaften sind die eines unendlich häufig genauen Fehlerdetektors, wie ihn Garg und Mitchell definieren [70]. Die letzten drei Eigenschaften definieren die Beziehung der Verdächtigungen/Rehabilitierungen zu den anderen Ereignissen, die auf einem entfernten Prozeß stattfinden.

Abbildung 4.3 zeigt den Pseudocode eines kausalen unendlich häufig genauen Fehlerdetektors gemäß Definition 4.1. Der Hauptvorteil dieses Fehlerdetektors ist, daß man ihn in asynchronen Systemen implementieren kann. Der Vektor *ac* ist eine Vektoruhr, die anzeigt, wieviele Ereignisse pro Prozeß bereits dem Monitor bekannt sind. Die Projektion $proj_i(ac)$ auf den i-ten Wert wird benutzt, um die Verdächtigungs- und Rehabilitierungsereignisse in den Ereigniskontext auf den Prozessen einzubetten. Die Eigenschaften "maximal eine Verdächtigung" und "maximal eine Rehabilitierung" werden durch zusätzliche Felder erreicht, welche die jeweils zuletzt getätigte Verdächtigung/Rehabilitierung pro Prozeß speichern. Der Algorithmus zur Implementierung des Fehlerdetektors entstammt einem Artikel von Garg und Mitchell [70]. Der Nachweis, daß der Fehlerdetektor tatsächlich unendlich oft genau ist, ist an anderer Stelle bereits durchgeführt worden [71]. Im folgenden wird davon ausgegangen, daß Fehlerdetektoren mit den genannten Eigenschaften zur Verfügung stehen.

4.4 Die Beobachtungsmodalitäten *negotiably* und *discernibly*

Wie oben ausgeführt, sind die Beobachtungsmodalitäten *possibly* und *definitely* in fehlerbehafteten, asynchronen Systemen nicht mehr beobachterinvariant. In diesem Abschnitt geht es um die Frage, welche alternativen Modalitätsdefinition es in diesem Systemmodell geben kann.

Schuld an der Beobachterabhängigkeit der gegebenen Modalitäten ist die Unzuverlässigkeit der Fehlerdetektoren, ein Nachteil, den man im gegebenen Systemmodell nicht umgehen kann. Eine Möglichkeit, beobachterunabhängige Modalitäten zu schaffen, basiert darum auf einer einheitlichen "globalen" Fehlerdetektorsemantik. Ein solcher Fehlerdetektor, den man durch einen geeigneten

```
1  On every application process p_i:

2      periodically do
3          send "alive" to ⟨all monitors⟩

4  On every monitor process b_j:

5    variables:
6      ac array [1..n] of ℕ init (0, ..., 0)                {* application clock *}
7                                          {* keeps current vector time of monitor *}
8      suspects set of ⟨processes⟩ init ∅                        {* suspect list *}
9      timeout array [1..n] of ℕ init (c, ..., c)
10     watch array [1..n] of ⟨timer⟩ init (c, ..., c)          {* process timers *}
11     last_suspicion array [1..n] of ℕ init (0, ..., 0)
12     last_rehabilitation array [1..n] of ℕ init (0, ..., 0)

13   algorithm:
14     upon ⟨receiving "alive" or a control message from p_i⟩ do
15       if p_i ∈ suspects then
16         suspects := suspects \ {p_i}
17         timeout[i] := timeout[i] + 1
18         if proj_i(ac) > last_rehabilitation[i] then
19           ⟨notify application that "b_j rehabilitates p_i after proj_i(ac)"⟩
20           last_rehabilitation := proj_i(ac)
21         endif
22       endif
23       ⟨reset watch[i] to timeout[i]⟩
24     upon ⟨expiry of watch[i]⟩ do
25       if p_i ∉ suspects then
26         suspects := suspects ∪ {p_i}
27         if proj_i(ac) > last_suspicion[i] then
28           ⟨notify application that "b_j suspects p_i after proj_i(ac)"⟩
29           last_suspicion := proj_i(ac)
30         endif
31       endif
```

Abbildung 4.3: Implementierung eines Fehlerdetektors gemäß Definition 4.1 in asynchronen Systemen. Dieser Fehlerdetektor ordnet zusätzlich die eigenen Meldungen in den Ereigniskontext auf den verdächtigten Prozessen ein. Der Wert c ist eine beliebige positive Konstante.

verteilten Algorithmus realisieren muß, generiert dieselben Verdächtigungs- und Rehabilitierungsereignisse auf allen Beobachtern. In diesem Fall sagen wir, Prozeß p_i *wird verdächtigt (bzw. rehabilitiert) nach dem Ereignis* e_k^i (man beachte, daß diese Ereignisse keinen Bezug mehr zu einem speziellen Beobachter haben). Wenn alle Beobachter dieselben Fehlerdetektorinformationen bekommen, dann haben sie alle die gleiche Sicht auf den Verband konsistenter Zustände. In diesem Verband lassen sich dann wieder beobachterunabhängige Modalitäten definieren.

Es gibt prinzipiell zwei Möglichkeiten der Definition einer globalen Fehlerdetektorsemantik, die hier als *pessimistisch* und *optimistisch* bezeichnet werden.

4.2 Definition (pessimistischer Fehlerdetektor) Ein *pessimistischer Fehlerdetektor* ist ein Fehlerdetektor, der die folgenden beiden Bedingungen erfüllt:

- (Sicherheit) p_i wird genau dann nach Ereignis e_k^i verdächtigt (bzw. rehabilitiert), wenn es einen Beobachter b_r gibt, so daß b_r Prozeß p_i nach Ereignis e_k^i verdächtigt (bzw. rehabilitiert).
- (Lebendigkeit) Wenn es einen Beobachter b_r gibt, so daß b_r den Prozeß p_i nach e_k^i verdächtigt (bzw. rehabilitiert), dann wird p_i nach endlicher Zeit nach e_k^i verdächtigt (bzw. rehabilitiert).

4.3 Definition (optimistischer Fehlerdetektor) Ein *optimistischer Fehlerdetektor* ist ein Fehlerdetektor, der die folgenden beiden Bedingungen erfüllt:

- (Sicherheit) p_i wird genau dann nach Ereignis e_k^i verdächtigt (bzw. rehabilitiert), wenn für alle Beobachter b_r gilt, daß b_r den Prozeß p_i nach Ereignis e_k^i verdächtigt (bzw. rehabilitiert).
- (Lebendigkeit) Wenn für alle Beobachter b_r gilt, daß b_r den Prozeß p_i nach e_k^i verdächtigt (bzw. rehabilitiert), dann wird p_i nach endlicher Zeit nach e_k^i verdächtigt (bzw. rehabilitiert).

Ein optimistischer Fehlerdetektor wartet, bis alle lokalen Fehlerdetektormodule einen bestimmten Prozeß verdächtigen, bevor er selbst eine Verdächtigung ausspricht. Dies wird “optimistisch” genannt in Analogie zu sogenannten optimistischen Netzwerkprotokollen. Dies sind Protokolle, die ohne irgendwelche Koordination bestimmte Aktionen ausführen in der Hoffnung, daß diese Koordination nicht nötig sein wird. Beispiele sind optimistische *update*-Protokolle in replizierten Datenbanken, die eine gewünschte Aktualisierung ausführen und eventuelle Versionskonflikte später mit besonderen Resolutionsalgorithmen behandeln. In diesem Sinne ignoriert ein optimistischer Fehlerdetektor sporadische Verdächtigungen in der Hoffnung, daß sie sich als falsch herausstellen. Im Gegensatz dazu meldet ein pessimistischer Fehlerdetektor sich, sobald auch nur ein

einzelner lokaler Fehlerdetektor einen bestimmten Prozeß verdächtigt. Im Sinne eines pessimistischen Beobachters kann man also sofort eine Aktion ausführen, sobald auch nur die geringste Andeutung eines Prozeßabsturzes aufkommt.

Wie im Falle der Fehlerdetektormodule macht es Sinn zu fordern, daß optimistische und pessimistische Fehlerdetektoren maximal eine Verdächtigung (bzw. Rehabilitierung) zwischen zwei normalen Ereignissen auf einem Prozeß aussprechen und daß zwischen einer Verdächtigung und der anschließenden Rehabilitierung keine normalen Ereignisse auf dem verdächtigten Prozeß stattfinden.

4.4 Definition (zusätzliche Fehlerdetektoreigenschaften) Ein optimistischer oder pessimistischer Fehlerdetektor habe zusätzlich die folgenden beiden Eigenschaften:

- (maximal eine Verdächtigung) Prozeß p_i wird nach e_k^i maximal ein Mal verdächtigt.
- (maximal eine Rehabilitierung) Prozeß p_i wird nach e_k^i maximal ein Mal rehabilitiert und auch nur, wenn p_i zuvor nach e_k^i verdächtigt wurde.
- (garantierte Totzeit) Wenn p_i nach e_k^i verdächtigt wird und ein Ereignis e_{k+1}^i existiert, dann wird p_i nach e_k^i rehabilitiert und zwar vor dem Ereignis e_{k+1}^i

Die (globalen) Verdächtigungen und Rehabilitierungen eines optimistischen oder pessimistischen Fehlerdetektors stellen Ereignisse dar, die quasi gleichwertig zu normalen Ereignissen auf Applikationsprozessen sind. Sie werden wie folgt in die kausale Ordnung eingebettet: Wenn s_k^i und r_k^i die Verdächtigungs- bzw. Rehabilitierungsereignisse über p_i nach e_k^i darstellen, dann gilt $e_k^i \to s_k^i \to r_k^i \to e_{k+1}^i$. Wie oben bereits beschrieben, werden die Fehlerdetektorereignisse wie Zustandsänderungen auf einem entfernten Prozeß behandelt. Da es sich bei Fehlerdetektorereignissen quasi um lokale Zustandsänderungen handelt, ist jeder Zustand, der durch eine Änderung der *up*-Variable aus einem konsistenten Zustand hervorgeht, wieder konsistent. Das gilt auch für zweimaliges Ändern von *up*, d.h. eine Verdächtigung mit anschließender Rehabilitierung: Nach der Rehabilitierung befindet sich das System im gleichen (konsistenten) Zustand wie vor der Verdächtigung. Abbildung 4.4 zeigt eine verteilte Berechnung mit globalen Verdächtigungen und Rehabilitierungen sowohl als Raum/Zeit-Diagramm (oben) als auch in Form des Verbandes konsistenter Zustände (unten, der Verband konsistenter Zustände ist grau hinterlegt). Zusammen mit der Kausalordnung, in die die Fehlerdetektorereignisse eingebettet worden sind, bildet also die so erweiterte Menge der konsistenten Zustände wieder einen Verband, der *Verband über dem erweiterten Zustandsraum* genannt werden soll. Die entsprechenden *inf*- und *sup*-Operationen sind wieder der Schnitt und die Vereinigung der den Zuständen entsprechenden

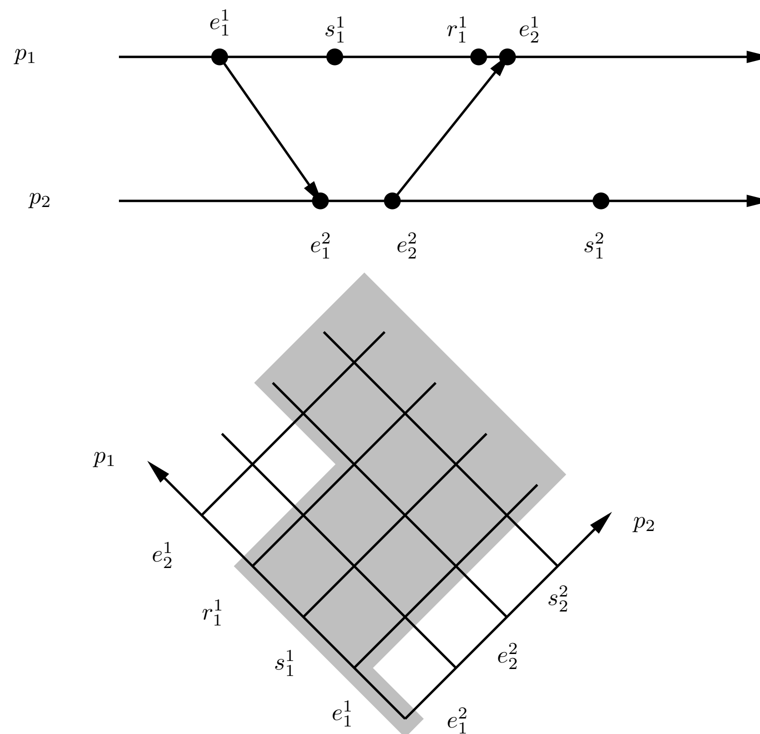


Abbildung 4.4: Eine verteilte Berechnung mit globalen Verdächtigungen und Rehabilitierungen und dem entsprechenden Zustandsverband.

Ereignismengen.

Basierend auf den beiden unterschiedlichen Arten, wie die globalen Verdächtigungen und Rehabilitierungen produziert werden können, kann man zwei verschiedene Varianten des "erweiterten" Zustandsverbandes definieren, eine optimistische und eine pessimistische. Sie entsprechen jeweils dem Schnitt und der Vereinigung der einzelnen Zustandsverbände auf den Monitoren.

4.5 Definition (optimistischer Zustandsverband einer Berechnung) Ein Verband über dem erweiterten Zustandsraum einer Berechnung heißt genau dann *optimistisch*, wenn die in seiner Definition verwendeten Verdächtigungs- und Rehabilitationsereignisse von einem optimistischen Fehlerdetektor stammen.

4.6 Definition (pessimistischer Zustandsverband einer Berechnung) Ein Verband über dem erweiterten Zustandsraum einer Berechnung heißt genau dann *pessimistisch*, wenn die in seiner Definition verwendeten Verdächtigungs- und Rehabilitationsereignisse von einem pessimistischen Fehlerdetektor stammen.

Mittels der Definitionen 4.5 und 4.6 kann man dann die folgenden beiden neuen Beobachtungsmodalitäten festlegen.

4.7 Definition (*negotiably*(φ)) Sei φ ein globales Prädikat auf dem erweiterten Zustandsraum. Das Prädikat *negotiably*(φ) gilt genau dann für eine verteilte Berechnung, wenn *possibly*(φ) im pessimistischen Zustandsverband dieser Berechnung gilt.

4.8 Definition (*discernibly*(φ)) Sei φ ein globales Prädikat auf dem erweiterten Zustandsraum. Das Prädikat *discernibly*(φ) gilt genau dann für eine verteilte Berechnung, wenn *definitely*(φ) im optimistischen Zustandsverband der Berechnung gilt.

Das Wort *negotiable* bedeutet im Deutschen "verhandelbar". Die Namensgebung der neuen Modalität soll ausdrücken, daß man für ein gewisses Ereignis bereits Indizien gesammelt hat, aber noch darüber verhandeln muß, ob dieses Ereignis auch zutrifft. Von der Definition her erbt *negotiably* den "existentiellen" Charakter von *possibly*: *negotiably*(φ) gilt, wenn ein Zustandsverband L existiert, so daß in L eine Beobachtung Σ existiert, so daß in Σ ein Zustand S existiert, für den φ gilt. Die Modalität *negotiably* hat demnach dieselbe Affinität zu bestimmten Formen von Sicherheitseigenschaften wie *possibly* und eignet sich insbesondere zur Überprüfung von Fehlerannahmen. Beispielsweise kann man φ definieren als "maximal $\lfloor n/2 \rfloor$ Prozesse stürzen ab". Wenn sich während der Überprüfung einer Berechnung herausstellt, daß *negotiably*$(\neg\varphi)$ gilt, dann gibt es eine mögliche Beobachtung, die die Verletzung der Fehlerannahme feststellen kann. Dies

kann zum Anlaß genommen werden, die Güte der Fehlerannahme nochmal zu überdenken.

Die Modalität *discernibly* läßt sich hingegen einfach verstehen in Analogie zu *definitely*. Hier dominiert der Allquantor: Für alle möglichen Zustandsverbände L gilt, daß für alle möglichen Beobachtungen Σ in L gilt, daß ein Zustand S in Σ existiert, für den φ gilt. Im Deutschen bedeutet das Wort *discernible* "unterscheidbar", "erkennbar" oder "sichtbar". Der Name soll ausdrücken, daß ein Prädikat, welches *discernible* ist, (relativ) eindeutig erkennbar zugetroffen hat. Entsprechend besteht zwischen derartigen Prädikaten und bestimmten Lebendigkeitseigenschaften eine besondere Beziehung: Wenn eine Berechnung die Eigenschaft $\Diamond\varphi$ erfüllen sollte, dann genügt die Entdeckung von *discernibly*(φ), um dies für eine konkrete Berechnung zu verifizieren. Ein Beispiel für φ wäre: "Jeder Prozeß erreicht entweder einen besonderen Terminierungszustand oder er stürzt ab."

Sonderfall 1: Keine Fehler treten auf. Bei der Diskussion der neuen Prädikatstransformatoren sind ein paar interessante Sonderfälle zu beachten. Der erste Fall ist von Bedeutung, wenn in einer Berechnung keine Fehler passieren. Wenn keine Fehler auftreten, dann besteht kein Unterschied zwischen den optimistischen bzw. pessimistischen Zustandsverbänden und dem originalen Zustandsverband des fehlerfreien Systems, da keinerlei zusätzliche Ereignisse stattfinden. Darum hat in diesem Fall *negotiably* dieselbe Semantik wie *possibly* und *discernibly* dieselbe Semantik wie *definitely*.

Sonderfall 2: perfekte Fehlerdetektoren und die Gültigkeit von φ. Der zweite Sonderfall tritt ein, wenn die verwendeten Fehlerdetektoren keine "Fehler" machen. Angenommen, wir hätten perfekte Fehlerdetektoren zur Verfügung, die auch erst ihre Verdächtigung ausrufen, nachdem die letzte Nachricht vom abgestürzten Prozeß empfangen worden ist, dann besteht immer ein kausaler Zusammenhang zwischen dem Prozeßabsturz und der Verdächtigung. Wie in Abbildung 4.5 kann man im Raum/Zeit-Diagramm dazu eine "virtuelle" Nachricht eintragen. Da die einzelnen Fehlerdetektormodule alle das gleiche Verhalten aufweisen, wären der optimistische und der pessimistische Zustandsverband dann identisch. Wenn ein Algorithmus in diesem Szenario *negotiably*(φ) entdeckt, kann man mit Fug und Recht behaupten, daß φ gegolten haben könnte (im Sinne von *possibly*). Bei unzuverlässigen Fehlerdetektoren muß man notwendigerweise davon ausgehen, daß φ nie in Wirklichkeit gegolten haben könnte, weil ein Fehlerdetektor eine falsche Verdächtigung gemacht hat.

Sonderfall 3: perfekte Fehlerdetektoren und Absturzprädikate. Der dritte Sonderfall betrachtet wieder perfekte Fehlerdetektoren (wie in Fall 2) sowie Prädikate der Form $\varphi \equiv \neg up[i]$. Bei perfekten Fehlerdetektoren sind der optimi-

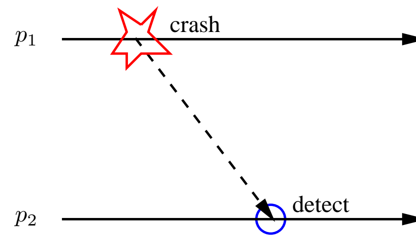


Abbildung 4.5: Virtuelle Absturznachricht bei perfekten Fehlerdetektoren.

stische und der pessimistische Zustandsverband wie oben besprochen identisch. Außerdem wird es niemals Rehabilitierungen geben, da niemals ein Prozeß zu Unrecht verdächtigt wird. Das Prädikat φ ist somit ein *stabiles Prädikat*, d.h. ein Prädikat, welches nie mehr falsch wird, wenn es einmal gilt. Für stabile Prädikate gibt es keinen Unterschied zwischen *possibly* und *definitely*. Entsprechend wird für das hier gewählte φ die Modalität *negotiably*(φ) genau dann entdeckt, wenn *discernibly*(φ) erkannt wird. Dies unterstreicht, daß die Modalitäten *negotiably* und *discernibly* erst Sinn machen, wenn Fehlerdetektoren unzuverlässig sind.

4.5 Entdeckungsalgorithmen I

Die beiden Beobachtungsmodalitäten *negotiably* und *discernibly* würden wenig Sinn machen, wenn sie in asynchronen Systemen mit *crash*-Fehlern nicht implementiert werden könnten. In den folgenden Abschnitten wird gezeigt, daß Algorithmen existieren, welche das Beobachtungsproblem für die betreffende Modalität lösen. Aus didaktischen Gründen erfolgt die Präsentation in zwei Teilen: Zunächst wird angenommen, daß im Gegensatz zu den Applikationsprozessen keiner der Monitorprozesse abstürzen kann. Unter dieser Annahme können die Grundprinzipien der Entdeckungsalgorithmen einfacher dargestellt werden. Im Abschnitt 4.6 wird diese Annahme weiter abgeschwächt. Dort werden Algorithmen vorgestellt, die auch dann korrekt sind, wenn eine bestimmte Anzahl von Monitorprozessen abstürzen dürfen.

4.5.1 Zwei logische Uhren und ihre Zeitstempel

Modularität des Algorithmus. Der Kern der Entdeckungsalgorithmen ist die Realisierung eines globalen Fehlerdetektors. Dieser Fehlerdetektor generiert aus den Informationen der einzelnen Fehlerdetektormodule eine “globale” Sicht auf die Fehlerdetektorereignisse. Aus dieser Sicht kann man optimistische bzw. pessimistische Versionen des Zustandsverbandes konstruieren, worauf man

dann einen der publizierten Standardalgorithmen für die Entdeckung von *possibly* oder *definitely* anwenden kann.

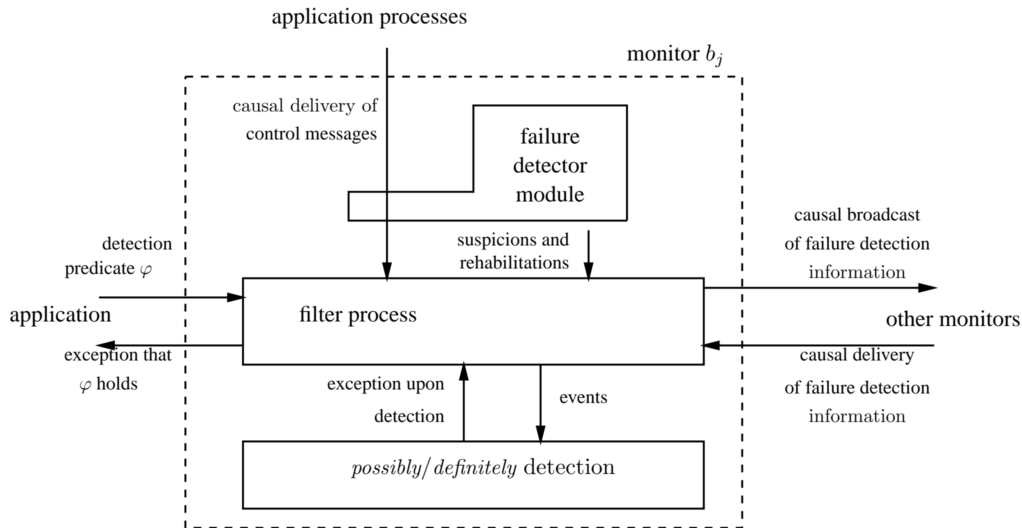


Abbildung 4.6: Architektur des Filters für einen globalen Fehlerdetektor. Die leicht asymmetrische Form des Fehlerdetektormoduls geht auf die Notwendigkeit zurück, die speziellen Fehlerdetektoreigenschaften implementieren zu müssen.

Filterfunktion des Algorithmus. Der globale Fehlerdetektor wirkt wie ein Filter, der die eingehenden Kontrollnachrichten und Fehlerdetektorereignisse aufnimmt, intern verarbeitet und an ein externes Modul weiterleitet, in dem die eigentliche *possibly*- oder *definitely*-Entdeckung abläuft. Die Architektur des Filters ist in Abbildung 4.6 dargestellt. Das Problem beim Bau eines globalen Fehlerdetektors ist natürlich, daß alle Beobachter dieselben Verdächtigungen bzw. Rehabilitierungen quasi zur selben Zeit erhalten müssen.

Applikations- und Monitoruhr. Wann immer ein lokales Fehlerdetektormodul einen entfernten Prozeß verdächtigt oder rehabilitiert, wird eine spezielle Kontrollnachricht an alle anderen Monitorprozesse geschickt. Diese Nachricht wird mit zwei speziellen Zeitstempeln versehen, die eine spätere Einordnung des Ereignisses in den Zustandsverband ermöglichen. Zu diesem Zweck benutzt jeder Monitor zwei logische Uhren:

1. Die *Applikationsuhr* ist eine Vektoruhr der Größe n, mit deren Hilfe die kausal-konsistente Zustellung der regulären Kontrollnachrichten organisiert wird.

2. Die *Monitoruhr* ist eine Vektoruhr der Größe m, mit deren Hilfe der kausale *broadcast* derjenigen Kontrollnachrichten erreicht wird, die von lokalen Fehlerdetektormodulen stammen.

Wann immer ein Monitor b_j einen entfernten Prozeß p_i verdächtigt oder rehabilitiert, wird also eine Nachricht der Form $((i, j, ats, x), mts)$ an alle anderen Monitore geschickt. Hierbei ist mts der aktuelle Wert der Monitoruhr, ats der aktuelle Wert der Applikationsuhr und $x =$'s' im Falle einer Verdächtigung und $x =$ 'r' im Falle einer Rehabilitierung. Der Wert von mts wird ausschließlich dazu verwendet, um eine kausale Ordnung dieser Fehlerdetektornachrichten beim Empfang zu erreichen, und wird daher bei den folgenden Ausführungen manchmal der Einfachheit halber weggelassen. Mit Hilfe des Wertes ats der Applikationsuhr ist es möglich, das Fehlerdetektorereignis in Beziehung zu setzen zu dem letzten Ereignis, welches dem Beobachter b_j von Prozeß p_i bekannt ist. Die Nummer dieses Ereignisses ist gerade der i-te Eintrag im Vektor ats. Formal bezeichne die Funktion $proj_i(v)$ die Projektion des Vektors v auf seine i-te Komponente. Dann bedeutet also der Erhalt der Nachricht (i, j, ats, x), daß Monitor b_j den Prozeß p_i nach Ereignis $e^i_{proj_i(ats)}$ verdächtigte bzw. rehabilitierte.

Wann sind alle Verdächtigungen von einem Beobachter eingetroffen? Ein Problem bei der Konstruktion eines globalen Zustandsverbandes ist folgendes: Ein Monitor weiß lokal nicht, ob und ggf. wann ein anderer Monitor einen entfernten Prozeß verdächtigte. Die kausale Ordnung auf den verschickten Fehlerdetektornachrichten schafft hier Abhilfe und ist Grundlage für einen wichtigen Sachverhalt, der im folgenden Lemma präzisiert wird.

4.9 Lemma Wenn zwei Fehlerdetektorkontrollnachrichten m_1 und m_2 mit $m_1 = ((i, j, ats_1, x_1), mts_1)$ und $m_2 = ((i, j, ats_2, x_2), mts_2)$ an einen Monitor ausgeliefert werden und wenn gilt $proj_i(ats_1) < proj_i(ats_2)$ dann wird m_1 immer vor m_2 ausgeliefert.

Beweis

BEWEISIDEE: Zentral ist die folgende Implikation: Wenn $proj_i(ats_1) < proj_i(ats_2)$, dann muß $mts_1 < mts_2$ gelten. Den Rest erledigt die kausale Ordnung bei der Zustellung der Fehlerdetektornachrichten. (Hier steht $<$ für die kleiner-Relation auf Vektoren, d.h. $v_1 < v_2$ genau dann, wenn für alle Einträge gilt $v_1[i] \leq v_2[i]$ und für mindestens einen Eintrag gilt $v_1[i] < v_2[i]$.)

1 $\langle 1 \rangle$1. Seien also $m_1 = ((i, j, ats_1, x_1), mts_1)$ und $m_2 = ((i, j, ats_2, x_2), mts_2)$. Es handelt es sich beide Male um eine Fehlerdetektornachricht von Monitor b_j, der Prozeß p_i nach dem Ereignis $e' = e^i_{proj_i(ats_1)}$ bzw. $e'' = e^i_{proj_i(ats_2)}$ verdächtigt bzw. rehabilitiert.

Beweis: Folgt aus der Definition der Fehlerdetektornachrichten und der Tatsache, daß i und j bei beiden Nachrichten gleich sind. □

2 ⟨1⟩2. Bei e' und e'' handelt es sich um zwei verschiedene Ereignisse (d.h. $e' \neq e''$) für die $e' \rightarrow e''$ gilt.

Beweis: Folgt aus Schritt ⟨1⟩1 und der Voraussetzung $proj_i(ats_1) < proj_i(ats_2)$. □

3 ⟨1⟩3. Die Kontrollnachricht über e' wird vor der Kontrollnachricht über e'' bei b_j zugestellt.

Beweis: Folgt aus Schritt ⟨1⟩2 und der kausalen Zustellung der Kontrollnachrichten. □

4 ⟨1⟩4. Das Ereignis, das m_1 auf b_j auslöst, passiert vor dem Ereignis, das m_2 auf b_j auslöst.

Beweis: Folgt aus Schritt ⟨1⟩3. □

5 ⟨1⟩5. Für die Zeitstempel mts_1 und mts_2 gilt: $mts_1 < mts_2$.

Beweis: Folgt aus Schritt ⟨1⟩4 und der Art und Weise, wie die Monitorzeitstempel konstruiert werden. □

6 ⟨1⟩6. Q.E.D.

Beweis: Die Konklusion des Lemmas folgt aus der kausalen Zustellung der Fehlerdetektorkontrollnachrichten. □

Wenn man sich in die Lage eines Monitors versetzt, kann man die Bedeutung von Lemma 4.9 besser verstehen. Angenommen, es gibt drei Monitore b_1, b_2 und b_3, die eine verteilte Berechnung mit zwei Prozessen p_1 und p_2 beobachten. Jeder Monitor macht sich bei der Konstruktion des Verbandes der konsistenten Zustände ein Bild von den Ereignissen, die jeweils auf den Prozessen passiert sind (eingehende Fehlerdetektornachrichten werden wie normale Prozeßereignisse interpretiert). Wenn Beobachter b_1 beispielsweise eine Fehlerdetektornachricht $(1, 2, (3, 2), s)$ erhält, bedeutet dies: Monitor b_2 verdächtigte p_1 nach Ereignis $e^1_{proj_1((3,2))} = e^1_3$. Lemma 4.9 garantiert, daß in Zukunft keinerlei Fehlerdetektornachrichten von b_2 mehr eintreffen werden, in denen p_1 *vor* dem Ereignis e^1_3 verdächtigt oder rehabilitiert wurde. Das bedeutet, bezüglich b_2 wird sich die Ereignisfolge auf p_1, die bis zum Ereignis e^1_3 reicht, nicht mehr ändern.

4.5.2 Der stabile Bereich

Den in Lemma 4.9 beschriebenen Sachverhalt kann man für eine Menge von Monitoren verallgemeinern. Angenommen, im obigen Beispiel habe die empfangene Nachricht den Monitorzeitstempel $mts = (1, 1, 1)$. Das bedeutet, daß b_1 inzwischen von jedem anderen Monitor mindestens eine Fehlerdetektornachricht erhalten hat und daß keine solche Nachrichten mehr ankommen werden, welche dieser Nachricht kausal vorausgehen. Die drei Fehlerdetektornachrichten seien mit m_1,

m_2 und m_3 bezeichnet und die Applikationszeitstempel dieser Nachrichten seien $(3,1)$, $(3,2)$ und $(2,3)$.

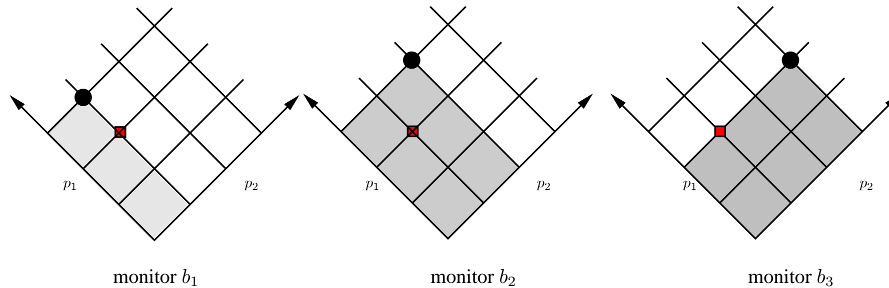


Abbildung 4.7: Ereignisse, die den drei Applikationszeitstempeln $(3,1)$, $(3,2)$ und $(2,3)$ entsprechen. Das kleine Quadrat deutet an, welche Ereignisse bereits von allen Monitoren beobachtet worden sind.

Abbildung 4.7 zeigt die konsistenten Zustände, welche den angegebenen Zeitstempeln entsprechen, innerhalb der Zustandsverbände der einzelnen Monitore. Welche Ereignisse haben bereits alle Monitore beobachtet? Die Antwort auf diese Frage ist in Abbildung 4.7 durch ein kleines Quadrat bezeichnet: Es sind alle Ereignisse, die unterhalb von Zustand mit den Koordinaten $(2,1)$ liegen. Dieser Zustand entspricht dem Schnitt hinter den Ereignissen e_2^1 und e_1^2. Lemma 4.10 besagt nun, daß auch keinerlei Fehlerdetektornachrichten mehr eintreffen werden, die kausal vor Ereignis $(2,1)$ liegen. Demnach wird sich die durch den Zeitstempel $(2,1)$ markierte Region in Zukunft nicht mehr ändern und wird deshalb als *stabiler Bereich* (*settled region*) bezeichnet. Formal ist dies der Schnitt aller durch die aktuellen Applikationszeitstempel definierten Unterverbände. Algorithmisch berechnet sich der stabile Bereich aus dem komponentenweisen Minimum der Applikationszeitstempel der jeweils letzten Fehlerdetektornachrichten von jedem Monitor.

4.10 Lemma Sei

1. b_r ein beliebiger Monitor,

2. bezeichne $most_recent_ats[1..m, 1..n]$ ein Feld, in dem die Applikationszeitstempel der zuletzt empfangenen Fehlerdetektornachrichten der einzelnen Monitore gespeichert sind, und

3. definiere einen n-dimensionalen Vektor *common* als

$$common := \min_{l=1,\dots,m} \{most_recent_ats[l]\}$$

Dann gilt: Für alle Fehlerdetektornachrichten (i, j, ats, x), die noch an b_r ausgeliefert werden, ist $ats \geq common$. (Hier ist mit $\geq$ die Operation auf Vektoren, d.h. $v_1 \geq v_2$ genau dann, wenn $v_1 > v_2$ oder $v_1 = v_2$.)

Beweis

BEWEISIDEE: Das Lemma ist eine Verallgemeinerung von Lemma 4.9. Die Gültigkeit basiert auf der Konstruktion von *common*.

1 ⟨1⟩1. Seien auf einem Monitor b_r die Variablen *most_recent_ats* und *common* wie im Lemma beschrieben definiert und seien die einzelnen Spalten der Matrix *most_recent_ats* bezeichnet mit

$$most_recent_ats = (ats_1, \ldots, ats_m)$$

Demnach ist ats_j der Wert der Applikationsuhr von b_j, als b_j diejenige Fehlerdetektornachricht an b_r geschickt hat, die b_r von b_j zuletzt empfangen hat. Diese letzten Nachrichten der einzelnen Beobachter seien mit $m_1, \ldots, m_m$ bezeichnet.

Angenommen, b_r empfängt eine Fehlerdetektornachricht $m = (i, j, ats, x)$ von b_j. Beim Vergleich von m mit der zuletzt von b_j empfangenen Fehlerdetektornachricht m_j gilt: m ist kausal nach m_j abgeschickt worden.

BEWEIS: Folgt aus der kausalen Zustellung der Fehlerdetektornachrichten (mittels der Monitoruhr). □

2 ⟨1⟩2. Weiterhin gilt für die Applikationszeitstempel der beiden Nachrichten m_j und m, daß $ats \geq ats_j$.

BEWEIS: Folgt aus Schritt ⟨1⟩1 und der Tatsache, daß sich die Applikationsuhr nie verringert. □

3 ⟨1⟩3. Es gilt für alle j: $ats_j \geq common$.

BEWEIS: Folgt aus der Konstruktion von *common*. □

4 ⟨1⟩4. $ats \geq common$

BEWEIS: Folgt aus Schritten ⟨1⟩2 und ⟨1⟩3 und der Transitivität von $\geq$. □

5 ⟨1⟩5. Q.E.D.

BEWEIS: Die Konklusion des Lemmas folgt direkt aus Schritt ⟨1⟩4. □

Lemma 4.10 ermöglicht folgende Vorgehensweise: Ein verteilter Algorithmus puffert alle eingehenden Kontrollnachrichten und bestimmt immer dann den Wert von *common*, sobald eine Fehlerdetektornachricht eingeht. Anschließend kann der Algorithmus alle Ereignisse aus dem stabilen Bereich an das *possibly*- oder *definitely*-Erkennungsmodul weiterleiten. Bei diesem Vorgehen können gleichzeitig die eingegangenen Fehlerdetektornachrichten in "normale" Applikationsereignisse umgewandelt werden. Hierbei muß allerdings die optimistische bzw. pessimistische Fehlerdetektorsemantik berücksichtigt werden. Im ersten Fall bedeutet das zum Beispiel, daß nur dann ein Ereignis "p_i wird nach e_k^i verdächtigt" an das Entdeckungsmodul weitergegeben wird, wenn von allen Monitoren b_j ein Ereignis "b_j verdächtigt p_i nach e_k^i" vorliegt.

Fortschreiten des stabilen Bereiches. Da der stabile Bereich über Zeitstempel von Fehlerdetektornachrichten definiert ist, kann dieser sich nur vergrößern, wenn derartige Nachrichten mit wachsenden Applikationszeitstempeln eintreffen. Wenn beispielsweise ein Monitor b_j nie irgendwelche Verdächtigungen ausspricht, dann kann sich der stabile Bereich wegen der Minimumsbildung nie über den initialen Zustand ausdehnen. Um die Lebendigkeit des Algorithmus zu gewährleisten, muß man für einen regelmäßigen Nachrichtenstrom, notfalls durch "Null"-Nachrichten, von den Fehlerdetektoren sorgen.

4.5.3 Algorithmus zur Entdeckung von *discernibly*

In Abbildung 4.8 ist der Pseudocode für einen Algorithmus abgebildet, der *discernibly* erkennt. Die Anweisungen des Algorithmus können in zwei Teile gegliedert werden:

- Der erste Teil ist zuständig für den kausalen *broadcast* der Ereignisse des lokalen Fehlerdetektors. Der kausale *broadcast* basiert auf den Zeitstempeln der Monitoruhr *mc*. Um die Lebendigkeit des Algorithmus zu garantieren wird zusätzlich noch periodisch eine "leere" Verdächtigung an alle Beobachter verschickt.
- Der zweite Teil ist der eigentliche Filter und spielt die Rolle des Mittlers zwischen den eingehenden Kontrollnachrichten und dem Modul zur Entdeckung von *definitely*. Normale Kontrollnachrichten werden in einer Menge *app_events* gepuffert, die Fehlerdetektornachrichten in einer Menge *fd_events*. Immer wenn Fehlerdetektornachrichten (auch leere) eingehen, wird der Wert von *common* neu berechnet und es werden die Ereignisse, die im dadurch bestimmten stabilen Bereich liegen, an das *definitely*-Entdeckungsmodul weitergeleitet. In Zeile 27 erfolgt die Extraktion der globalen Fehlerdetektorereignisse im optimistischen Sinne. Ein Ereignis wird nur dann von der Menge *fd_events* in die Menge *app_events* überführt, wenn entsprechende Verdächtigungen von allen anderen Monitoren vorliegen.

Der Kern des Korrektheitsarguments für den Algorithmus wird in den folgenden beiden Lemmas vorgestellt. Sie zeigen, daß die Ereignisse, die an das *definitely*-Entdeckungsmodul weitergegeben werden, tatsächlich den optimistischen Zustandsverband repräsentieren.

4.11 Lemma Für alle Ereignisse, die in Zeile 36 an den *definitely*-Entdeckungsalgorithmus weitergegeben werden, gilt:

S1 Die Ereignisse werden in kausaler Ordnung weitergegeben.

```
 1  On every monitor process b_j:

 2    variables:
 3      mc array [1..m] of ℕ init (0,...,0)                         {* monitor clock *}
 4      ac array [1..n] of ℕ init (0,...,0)                     {* application clock *}
 5      most_recent_ats array [1..m, 1..n] of ℕ init ((0,...,0),...,(0,...,0))
 6      app_events set of ⟨state change⟩ × ℕ^n init ∅
 7                                   {* buffer of application events with timestamps *}
 8      fd_events set of {1,...,n} × {1,...,m} × ℕ^n × {s,r} init ∅
 9                          {* (i,j,ats,x) ∈ fd_events means b_j that suspected (x = s) *}
10                                     {* or rehabilitated (x = r) p_i at b_j-time ats *}
11      common[1..n] of ℕ init (0,...,0)                               {* settled area *}

12    algorithm:
13                                          {* failure detector module interface part: *}
14      upon ⟨suspicion of p_i by failure detector module⟩ do
15        ⟨causally broadcast (i,j,ac,s) to all monitors using mc⟩
16      upon ⟨rehabilitation of p_i by failure detector module⟩ do
17        ⟨causally broadcast (i,j,ac,r) to all monitors using mc⟩
18      periodically do
19        ⟨causally broadcast (0,j,ac,∗) to all monitors using mc⟩
20                                                                    {* filter part: *}
21      upon ⟨causal delivery of (e,ats) from p_i using ac⟩ do
22        app_events := app_events ∪ (e,ats)
23      upon ⟨causal delivery of (i,j,ats,x) from b_j using mc⟩ do
24        if i ≠ 0 then
25          fd_events := fd_events ∪ (i,j,ats,x)
26                                                        {* extract optimistic data *}
27          if ∀k ∈ {1,...,m}.∃(i,k,ats',x) ∈ fd_events.proj_i(ats) = proj_i(ats') then
28            case x of
29              's' : app_events := app_events ∪ (up[i] := false, ats)
30              'r' : app_events := app_events ∪ (up[i] := true, ats)
31            endcase
32          endif
33        endif
34        most_recent_ats[j] := ats
35        common := min_{l=1,...,m}{most_recent_ats[l]}        {* determine settled area *}
36        ⟨release all events from app_events prior to common to definitely detection
37              module in causal order⟩
38      upon ⟨notification "definitely(φ) holds" from detection module⟩ do
39        notify "discernibly(φ) holds" at application
```

Abbildung 4.8: Algorithmus zur Entdeckung von *discernibly* für den Fall, daß kein Monitor abstürzt. Die Funktion $proj_i(x)$ ist die Projektion eines Vektors x auf seine i-te Komponente.

S2 Wenn das weitergegebene Ereignis ein Applikationsereignis ist, dann ist es ein Ereignis, was in der Basisberechnung stattgefunden hat.

S3 Verdächtigungen und Rehabilitierungen werden gemäß der Definition des optimistischen Fehlerdetektors weitergegeben, d.h. nur dann, wenn alle Monitore eine entsprechende Verdächtigung oder Rehabilitierung geäußert haben.

Beweis

1 $\langle 1 \rangle$1. Es gilt Eigenschaft S1.
Beweis: Die Mechanismen zur Erfüllung der Eigenschaft S1 sind in Abbildung 4.8 nicht explizit ausprogrammiert worden. Die Ordnung, in der die einzelnen Ereignisse weitergegeben werden müssen, wird durch die jeweils in *app_events* beigefügte Zeitmarke *ats* bestimmt. Man muß nur darauf achten, daß eventuelle Verdächtigungen und Rehabilitierungen (die möglicherweise denselben Zeitstempel haben wie das vor ihnen stattgefundene Ereignis) in der richtigen Weise in die Kausalordnung einsortiert werden. □

2 $\langle 1 \rangle$2. Es gilt Eigenschaft S2.
Beweis: In Abbildung 4.8 werden keinerlei neue Nachrichten generiert. Es werden nur diejenigen Applikationsereignisse in *app_events* aufgenommen, die in Zeile 21 an den Monitor ausgeliefert worden sind. Da Prozesse nur dann eine Kontrollnachricht schicken, wenn ein Ereignis passiert ist, folgt die Eigenschaft S2. □

3 $\langle 1 \rangle$3. Es gilt Eigenschaft S3.

3.1 $\langle 2 \rangle$1. In Zeile 27 wird nur dann ein Fehlerdetektorereignis generiert, wenn für alle $k \in \{1, \ldots, m\}$ eine Nachricht vom Typ (i, k, ats, x) vorliegt und die i-te Komponente von ats überall gleich ist.
Beweis: Folgt aus dem Algorithmus. □

3.2 $\langle 2 \rangle$2. Alle Monitore haben eine Fehlerdetektornachricht in Zeile 15 oder 17 abgeschickt.
Beweis: Wegen Schritt $\langle 2 \rangle$1 wurden in Zeile 23 insgesamt m nachrichten von verschiedenen Monitoren der entsprechenden Art empfangen. Wegen der Zuverlässigkeit der Nachrichtenkanäle haben demnach alle Monitore eine solche Nachricht in Zeile 15 oder 17 abgeschickt. □

3.3 $\langle 2 \rangle$3. Alle Monitore haben den Prozeß p_i zum Zeitpunkt $proj_i(ats)$ verdächtigt bzw. rehabilitiert.
Beweis: Folgt aus Schritt $\langle 2 \rangle$2 und dem Algorithmus. □

3.4 $\langle 2 \rangle$4. Q.E.D.
Beweis: Folgt direkt aus Schritt $\langle 2 \rangle$3. Die zusätzlichen Fehlerdetektoreigenschaften aus Definition 4.4 (maximal eine Verdächtigung/Rehabilitierung, garantierte Totzeit) folgen aus den entsprechenden Eigenschaften der lokalen Fehlerdetektoren. □

4 ⟨1⟩4. Q.E.D.
Beweis: Die Eigenschaften S1, S2 und S3 wurden in den Schritten ⟨1⟩1, ⟨1⟩2 und ⟨1⟩3 gezeigt. □

4.12 Lemma Für jedes Ereignis e, welches zum Zeitpunkt *ats* auf einem Prozeß p_i stattfindet, gilt: Nach endlicher Zeit gilt für den Wert von *common* auf jedem Monitor b_j: $common \geq ats$

Beweis

1 ⟨1⟩1. Sei e ein beliebiges Ereignis, welches zur Applikationszeit *ats* stattfindet. Eine Nachricht über dieses Ereignis wird nach endlicher Zeit bei allen Monitoren empfangen und in Zeile 21 zugestellt.
Beweis: Folgt aus der Tatsache, daß der Prozeß, auf dem e stattfindet, eine Kontrollnachricht (e, ats) an alle Beobachter sendet, sowie aus der Zuverlässigkeit der Kanäle. □

2 ⟨1⟩2. Auf jedem Monitor zeigt zum Zustellzeitpunkt dieser Nachricht zeigt die lokale Applikationsuhr mindestens auf den Wert *ats*.
Beweis: Folgt aus Schritt ⟨1⟩1 und der kausalen Zustellung der Nachrichten. □

3 ⟨1⟩3. Nach diesem Zeitpunkt wird nach endlicher Zeit in Zeile 19 eine "leere" Verdächtigung an alle Monitore verschickt, deren Applikationszeitstempel mindestens *ats* anzeigt.
Beweis: Folgt aus Schritt ⟨1⟩2 und dem Algorithmus. □

4 ⟨1⟩4. Nach endlicher Zeit wird diese Nachricht in Zeile 23 auf jedem Monitor zugestellt.
Beweis: Folgt aus der Zuverlässigkeit der Kanäle. □

5 ⟨1⟩5. Auf jedem Monitor gilt für alle k: $most_recent_ats[k] \geq ats$.
Beweis: Folgt aus Schritten ⟨1⟩3 und ⟨1⟩4, sowie dem Algorithmus. □

6 ⟨1⟩6. Q.E.D.
Beweis: Die Konklusion des Lemmas folgt aus Schritt ⟨1⟩5 und der Tatsache, daß *common* das Minimum aller Einträge in *most_recent_ats* ist. □

4.13 Lemma Falls alle Monitore einen Prozeß p_i nach einem Ereignis e_k^i verdächtigen/rehabilitieren oder falls in der Basisberechnung ein Ereignis stattfindet, wird ein entsprechendes Ereignis nach endlicher Zeit in Zeile 36 an das Erkennungsmodul weitergegeben.

Beweis

1 ⟨1⟩1. Die Konklusion des Lemmas gilt für alle Applikationsereignisse, die in der Basisberechnung stattfinden.
Beweis: Folgt aus Lemma 4.12 und dem Algorithmus. □

2 ⟨1⟩2. Die Konklusion des Lemmas gilt für alle Fehlerdetektorereignisse.

2.1 ⟨2⟩1. Wenn alle Monitore einen bestimmten Prozeß p_i nach einem Ereignis e_k^i verdächtigen bzw. rehabilitieren, schicken alle eine entsprechende Nachricht in den Zeilen 15 bzw. 17 an alle anderen Monitore.

BEWEIS: Folgt aus dem Algorithmus. □

2.2 ⟨2⟩2. Betrachte einen beliebigen Monitor b_j: Die Nachrichten aus Schritt ⟨2⟩1 werden nach endlicher Zeit in Zeile 23 zugestellt und anschließend in Zeile 25 in die Menge *fd_events* aufgenommen.

BEWEIS: Folgt aus der Zuverlässigkeit der Kanäle und dem Algorithmus. □

2.3 ⟨2⟩3. Nach endlicher Zeit wird dann in den Zeilen 27 bis 30 ein entsprechendes globales Ereignis mit einem Zeitstempel *ats* in *app_events* aufgenommen.

BEWEIS: Dies passiert Wenn das letzte der Ereignisse aus Schritt ⟨2⟩2 in die Menge *fd_events* aufgenommen wird. □

2.4 ⟨2⟩4. In der Basisberechnung hat es mindestens schon ein Ereignis e mit dem Applikationszeitstempel *ats* auf einem Prozeß gegeben.

BEWEIS: Folgt aus Schritt ⟨2⟩3. □

2.5 ⟨2⟩5. Auf jedem Monitor wird nach endlicher Zeit der Wert von *common* größer als der Wert von *ats*.

BEWEIS: Folgt aus Schritt ⟨2⟩4 und Lemma 4.12. □

2.6 ⟨2⟩6. Q.E.D.

BEWEIS: In dem durch Schritt ⟨2⟩5 definierten Zeitpunkt wird auf jedem Monitor das Fehlerdetektorereignis an das Detektionsmodul weitergegeben. □

3 ⟨1⟩3. Q.E.D.

BEWEIS: Schritte ⟨1⟩1 und ⟨1⟩2 decken alle Ereignistypen ab. □

4.14 Satz Sei φ ein Prädikat auf dem erweiterten Zustandsraum. Der Algorithmus aus Abbildung 4.8 löst das Beobachtungsproblem für *discernibly*(φ).

Beweis

1 ⟨1⟩1. Der Algorithmus aus Abbildung 4.8 erfüllt die Sicherheitsbedingung des Beobachtungsproblems für *discernibly*(φ).

BEWEIS: Folgt aus Lemma 4.11 und der Sicherheit des Entdeckungsalgorithmus für *definitely*. □

2 ⟨1⟩2. Der Algorithmus aus Abbildung 4.8 erfüllt die Lebendigkeitsbedingung des Beobachtungsproblems für *discernibly*(φ).

BEWEIS: Folgt aus Lemma 4.13 und der Lebendigkeit des Entdeckungsalgorithmus für *definitely*. □

3 ⟨1⟩3. Q.E.D.

BEWEIS: Folgt aus Schritten ⟨1⟩1 und ⟨1⟩2. □

4.5.4 Algorithmus zur Entdeckung von *negotiably*

Bis auf die Filteraktionen ist der Algorithmus zur Entdeckung von *negotiably* gleich dem Algorithmus aus Abbildung 4.8. Die Änderungen sind in Abbildung 4.9 dargestellt. Hier speichert die Menge *fd_events* nicht *alle* Fehlerdetektorereignisse wie bei der Entdeckung von *discernibly* sondern immer jeweils nur *ein* Ereignis der Form "b_j verdächtigte/rehabilitierte p_j nach e_k^i". Die Menge wird quasi als "Merker" benutzt, damit nicht mehrere Verdächtigungen/Rehabilitierungen der gleichen Art generiert werden.

```
1   On every monitor process b_j:

20                                                              {* filter part: *}
21      upon ⟨causal delivery of (e, ats) from p_i using ac⟩ do
22        app_events := app_events ∪ (e, ats)
23      upon ⟨causal delivery of (i, j, ats, x) from b_j using mc⟩ do
24        if i ≠ 0 then                                 {* extract pessimistic data *}
25          if ¬(∃(i, j, ats', x) ∈ fd_events.proj_i(ats) = proj_i(ats')) then
26            fd_events := fd_events ∪ (i, j, ats, x)
27            case x of
28              's' : app_events := app_events ∪ (up[i] := false, ats)
29              'r' : app_events := app_events ∪ (up[i] := true, ats)
30            endcase
31          endif
32        endif
33        most_recent_ats[j] := ats
34        common := min_{l=1,...,m}{most_recent_ats[l]}      {* determine settled area *}
35        ⟨release all events from app_events prior to common to possibly detection
36              module in causal order⟩
37      upon ⟨notification "possibly(φ) holds" from detection module⟩ do
38        notify "negotiably(φ) holds" at application
```

Abbildung 4.9: Pseudocode des Algorithmus zur Entdeckung von *negotiably*. Die Zeilen 2 bis 19 sind dieselben wie in Abbildung 4.8.

4.15 Lemma Für alle Ereignisse, die in Zeile 35 an den *possibly*-Entdeckungsalgorithmus weitergegeben werden, gilt:

S1 Die Ereignisse werden in kausaler Ordnung weitergegeben.

S2 Wenn das weitergegebene Ereignis ein Applikationsereignis ist, dann ist es ein Ereignis, was in der Basisberechnung stattgefunden hat.

S3 Verdächtigungen und Rehabilitierungen werden gemäß der Definition des pessimistischen Fehlerdetektors weitergegeben, d.h. nur dann, wenn min-

destens ein Monitor eine entsprechende Verdächtigung oder Rehabilitierung geäußert hat. In diesem Fall wird auch nur genau eine entsprechende Verdächtigung erzeugt.

Beweis

1 ⟨1⟩1. Es gelten Eigenschaften S1 und S2.
Beweis: Wie in Lemma 4.11. □

2 ⟨1⟩2. Es gilt Eigenschaft S3.

2.1 ⟨2⟩1. In Zeile 26 wird nur dann eine Verdächtigung/Rehabilitierung erzeugt, wenn noch kein Ereignis (i, j, ats', x) in *fd_events* vorliegt mit $proj_i(ats) = proj_j(ats')$.
Beweis: Folgt aus dem Algorithmus. □

2.2 ⟨2⟩2. Eine Nachricht der Form (i, j, ats, x) wurde in Zeile 23 empfangen.
Beweis: Folgt aus Schritt ⟨2⟩1 und dem Algorithmus. □

2.3 ⟨2⟩3. Eine Nachricht der Form (i, j, ats, x) wurde von einem Monitor b_j abgeschickt.
Beweis: Folgt aus Schritt ⟨2⟩2 und der Zuverlässigkeit der Kanäle. □

2.4 ⟨2⟩4. Monitor b_j verdächtigte p_i nach dem Ereignis mit der Nummer $proj_i(ats)$.
Beweis: Folgt aus Schritt ⟨2⟩3 und dem Algorithmus. □

2.5 ⟨2⟩5. Q.E.D.
Beweis: Folgt direkt aus Schritt ⟨2⟩4. □

3 ⟨1⟩3. Q.E.D.
Beweis: Folgt aus Schritten ⟨1⟩1 und ⟨1⟩2. □

Wegen der Ähnlichkeit in den Algorithmen gilt Lemma 4.12 auch für den Algorithmus aus Abbildung 4.9. Mit Hilfe dieses Lemmas läßt sich dann das folgende Lemma über die Lebendigkeit des Algorithmus beweisen.

4.16 Lemma Falls ein Monitor einen Prozeß p_i nach einem Ereignis e_k^i verdächtigt bzw. rehabilitiert oder falls in der Basisberechnung ein Ereignis stattfindet, wird ein entsprechendes Ereignis nach endlicher Zeit in Zeile 35 an das Erkennungsmodul weitergegeben.

Beweis

1 ⟨1⟩1. Für Applikationsereignisse der Basisberechnung gilt die Konklusion des Lemmas.
Beweis: Wie in Lemma 4.13. □

2 ⟨1⟩2. Für Fehlerdetektorereignisse gilt die Konklusion des Lemmas.

2.1 ⟨2⟩1. Angenommen, Monitor b_j verdächtigt/rehabilitiert p_i nach Ereignis e_k^i. Dann wird eine Nachricht (i, j, ats, x) an alle Monitore geschickt.
Beweis: Folgt aus dem Algorithmus. □

2.2 $\langle 2 \rangle 2$. Eine Nachricht vom Typ (i, j, ats, x) kommt bei allen Monitoren an.
BEWEIS: Folgt aus der Zuverlässigkeit der Kanäle. □

2.3 $\langle 2 \rangle 3$. Q.E.D.
BEWEIS: Falls noch kein entsprechendes Ereignis in *fd_events* vorliegt, wird es spätestens jetzt in diese Menge aufgenommen. Der Rest folgt aus der Tatsache, daß nach endlicher Zeit $common \geq ats$ wird (Lemma 4.12). □

3 $\langle 1 \rangle 3$. Q.E.D.
BEWEIS: Schritte $\langle 1 \rangle 1$ und $\langle 1 \rangle 2$ decken alle Fälle ab. □

4.17 Satz Sei φ ein Prädikat auf dem erweiterten Zustandsraum. Der Algorithmus aus Abbildung 4.9 löst das Entdeckungsproblem für *negotiably*(φ).

Beweis

1 $\langle 1 \rangle 1$. Der Algorithmus aus Abbildung 4.9 erfüllt die Sicherheitsbedingung des Beobachtungsproblems für *negotiably*(φ).
BEWEIS: Folgt aus Lemma 4.15 und der Sicherheit des Entdeckungsalgorithmus für *possibly*. □

2 $\langle 1 \rangle 2$. Der Algorithmus aus Abbildung 4.9 erfüllt die Lebendigkeitsbedingung des Beobachtungsproblems für *negotiably*(φ).
BEWEIS: Folgt aus Lemma 4.16 und der Lebendigkeit des Entdeckungsalgorithmus für *possibly*. □

3 $\langle 1 \rangle 3$. Q.E.D.
BEWEIS: Folgt aus Schritten $\langle 1 \rangle 1$ und $\langle 1 \rangle 2$. □

4.6 Entdeckungsalgorithmen II

Die Grundannahme bei der Konstruktion der Algorithmen in Abschnitt 4.5 war, daß Monitorprozesse im Gegensatz zu Applikationsprozessen nicht abstürzen. Diese Annahme ist aus theoretischer Sicht natürlich erlaubt, schränkt aber die Nützlichkeit der Algorithmen stark ein, denn oftmals laufen die Monitorprozesse auf denselben physikalischen Rechnerknoten wie Applikationsprozesse. In diesem Abschnitt werden Algorithmen vorgestellt, die auch dann noch korrekt sind, wenn maximal $t < m$ Monitorprozesse abstürzen.

4.6.1 Entdecken von *negotiably* bei Monitorabstürzen

Die Definition von *negotiably* macht offensichtlich auch dann noch Sinn, wenn Monitorprozesse abstürzen können. Beim Entwurf von Entdeckungsalgorithmen

muß man in diesem Szenario allerdings ein paar Kleinigkeiten beachten.

Es muß beispielsweise gewährleistet sein, daß jede getätigte Verdächtigung auch tatsächlich andere Monitore erreicht. Wenn ein Monitor einen Applikationsprozeß verdächtigt, aber sofort danach abstürzt, werden die anderen Monitor niemals von dieser Verdächtigung erfahren und unter Umständen ein Prädikat nicht entdecken, obwohl es in der gewählten Modalität galt. Als Lösung für dieses Problem muß man demnach fordern, daß die Code-Zeilen von der Verdächtigung bis zum Versenden der Kontrollnachricht im Algorithmus atomar ablaufen (d.h. entweder ganz oder gar nicht). Alternativ könnte man die Definition von *negotiably* ändern, indem man die Fehlerereignisse im pessimistischen Zustandsverband nur von *korrekten* Prozessen zuläßt. Dies würde aber eine *a priori* Entscheidung darüber voraussetzen, welche Prozesse abstürzen werden und welche nicht, eine Unmöglichkeit.

Lebendigkeit. Die Lebendigkeit des Algorithmus aus Abbildung 4.9 beruhte auf der Tatsache, daß immer wieder neu nach endlicher Zeit von jedem Monitor eine (unter Umständen leere) Fehlerdetektornachricht empfangen werden konnte. Wenn ein Monitor abstürzt, dann kann er keine Nachrichten mehr versenden und der Algorithmus würde auf den verbleibenden Monitoren stagnieren. Fehlende Nachrichten dürfen also nicht zu einer Blockade des Algorithmus führen.

Die Idee des Entdeckungsalgorithmus für *negotiably* unter der schwächeren Fehlerannahme ist die folgende: Monitore schicken wie in Abschnitt 4.5 mittels *broadcast* ihre Verdächtigungen an alle anderen. Diese Ereignisse werden ohne die Berechnung eines stabilen Bereiches sofort in reguläre Applikationsereignisse transformiert und direkt an das Entdeckungsmodul für *possibly* weitergeleitet. Dadurch wird jede Möglichkeit einer Blockierung ausgeschlossen und alle Ereignisse erreichen tatsächlich das Endeckungsmodul für *possibly*.

Bei der sofortigen Durchreichung der Fehlerdetektornachrichten können allerdings diese Nachrichten "zu spät" das *possibly*-Entdeckungsmodul erreichen, d.h. wenn Monitor b_j den Prozeß p_i nach Ereignis e^i_k verdächtigt, dann könnte dieses Ereignis das Entdeckungsmodul erreichen, nachdem bereits das Ereignis e^i_{k+1} an das Modul ausgeliefert wurde. Schließlich besteht ja kein "echter" kausaler Zusammenhang zwischen Fehlerdetektionsereignissen und den normalen Applikationsereignissen, den man durch Mechanismen wie Vektoruhren maßregeln könnte. Entdeckungsalgorithmen für *possibly* bauen den Zustandsverband normalerweise Schicht für Schicht auf und prüfen für jeden neu am äußeren Rand hinzugekommenen Zustand das gesuchte Prädikat. Bei verspäteten Ereignissen muß man diese in die "Berechnungsvergangenheit" eines Prozesses einsortieren, was einen "innerlich" veränderten Zustandsverband zur Folge hat. Wenn man diese Änderungen nicht in Betracht zieht, kann man die Gültigkeit von Prädikaten übersehen, wie das folgende Beispiel verdeutlicht.

Angenommen, ein Monitor b_1 hat bereits Nachricht von Ereignis e_3^1 auf Prozeß p_1 bekommen und erfährt nun, daß Monitor b_2 den Prozeß p_1 nach dem Ereignis e_2^1 verdächtigte. Das zu entdeckende Prädikat soll in diesem Beispiel *negotiably*(φ) für

$$\varphi \equiv \text{“}p_1 \text{ stürzte nach Ereignis } e_2^1 \text{ ab”}$$

lauten. Wenn man das verspätet eingegangene Ereignis ignoriert, dann würde man φ nicht entdecken, obwohl es tatsächlich galt. Die *possibly*-Entdeckung muß mit derartigen verspäteten Nachrichten also umgehen können. Im einfachsten Fall setzt man den Algorithmus, der für die *possibly*-Entdeckung zuständig ist, zurück in seinen Initialzustand und führt ihm die bisher weitergeleiteten Ereignisse plus die verspäteten wieder neu zu, jetzt allerdings in der richtigen zeitliche Reihenfolge. Wenn so etwas möglich ist, dann wird der Algorithmus zur Entdeckung von *possibly* nach endlicher Zeit *possibly*(φ) entdecken (wenn es galt) und somit wird der Gesamtalgorithmus auch *negotiably*(φ) entdecken. Die Lebendigkeit ist somit erreicht. (Natürlich muß man voraussetzen, daß die *possibly*-Erkennung vor der jeweils neuen Rücksetzung bis zum momentanen Berechnungszeitpunkt vorgedrungen ist.)

Sicherheit. Können verspätete Ereignisse zu einer Verletzung der Sicherheit führen? Dies wäre der Fall, wenn der Algorithmus voreilig ein Prädikat entdecken würde, was aber auf den zweiten Blick (d.h. wenn die verspäteten Informationen eingegangen sind) gar nicht gilt. Folgendes Lemma zeigt, daß dies glücklicherweise nicht passieren kann.

4.18 Lemma Sei L ein nichtleerer Zustandsverband einer Berechnung und sei L' der Zustandsverband, der durch Berücksichtigung von Fehlerdetektorereignissen entsteht, die auf mindestens einer Prozeßachse eingetragen werden. Dann existiert kein Prädikat φ über dem erweiterten Zustandsraum, für das *possibly*(φ) in L und $\neg$*possibly*(φ) in L' gilt.

Beweis

1 $\langle 1\rangle 1$. Sei L ein Zustandsverband für den *possibly*(φ) gilt. Das bedeutet, es existiert ein konsistenter Zustand S in L für den φ gilt.
Beweis: Folgt aus der Definition von *possibly*. □

2 $\langle 1\rangle 2$. L' geht aus L hervor durch Hinzufügung von Verdächtigungs- und Rehabilitationsereignissen auf mindestens einer Prozeßachse.
Beweis: Folgt aus der Konstruktion von L'. □

3 $\langle 1\rangle 3$. L' entsteht aus L durch Hinzufügung neuer Zustände (es werden keine Zustände aus L entfernt).
Beweis: Folgt aus Schritt $\langle 1\rangle 2$. □

4 $\langle 1\rangle 4$. In L' existiert ebenfalls ein Zustand S.
Beweis: Folgt aus Schritt $\langle 1\rangle 1$ und $\langle 1\rangle 3$. □

5 $\langle 1 \rangle 5.$ *possibly*(φ) gilt auch in L'.
BEWEIS: Folgt aus Schritt $\langle 1 \rangle 4$ und der Definition von *possibly*. □
6 $\langle 1 \rangle 6.$ Q.E.D.
BEWEIS: Folgt aus Schritt $\langle 1 \rangle 5$. □

```
1  On every monitor process b_j:

2    variables:
3      mc array [1..m] of ℕ init (0,...,0)                 {* monitor clock *}
4      ac array [1..n] of ℕ init (0,...,0)             {* application clock *}
5      fd_events set of {1,...,n} × {1,...,m} × ℕ^n init ∅

6    algorithm:
7                                  {* failure detector module interface part: *}
8      upon ⟨suspicion of p_i by failure detector module⟩ do
9        ⟨causally broadcast (i, j, ac, s) to all monitors using mc⟩
10     upon ⟨rehabilitation of p_i by failure detector module⟩ do
11       ⟨causally broadcast (i, j, ac, r) to all monitors using mc⟩
12                                                          {* filter part: *}
13     upon ⟨causal delivery of (e, ats) from p_i using ac⟩ do
14       ⟨release event (e, ats) to possibly detection module⟩
15     upon ⟨causal delivery of (i, j, ats, x) from b_j using mc⟩ do
16       if ¬(∃(i, j, ats', x) ∈ fd_events.proj_i(ats) = proj_i(ats')) then
17         fd_events := fd_events ∪ (i, j, ats, x)
18         case x of
19           's' : ⟨release event (up[i] := false, ats) to possibly detection module⟩
20           'r' : ⟨release event (up[i] := true, ats) to possibly detection module⟩
21         endcase
22         if proj_i(ats) < proj_i(ac) then
23           ⟨rollback possibly detection⟩
24         endif
25       endif
26     upon ⟨notification "possibly(φ) holds" from detection module⟩ do
27       notify "negotiably(φ) holds" at application
```

Abbildung 4.10: Pseudocode des Algorithmus zur Entdeckung von *negotiably*.

Abbildung 4.10 zeigt den Algorithmus zur Entdeckung von *negotiably* in Systemen, in denen auch Monitore abstürzen dürfen. Für die Sicherheit und Lebendigkeit des Algorithmus ist bereits in den vorangegangenen Abschnitten argumentiert worden. Die folgenden Lemmata bringen die Argumentation nochmals auf den Punkt.

4.19 Lemma Falls ein Monitor einen Prozeß p_i nach einem Ereignis e_k^i verdächtigt

bzw. rehabilitiert dann wird nach endlicher Zeit auf allen funktionsfähigen Monitoren ein entsprechendes Ereignis in Zeile 19/20 an das Erkennungsmodul für *possibly* weitergegeben. Falls in der Basisberechnung ein Ereignis stattfindet, wird ein entsprechendes Ereignis auf allen funktionsfähigen Monitoren nach endlicher Zeit in Zeile 14 an das Erkennungsmodul weitergegeben.

Beweis

1 $\langle 1 \rangle 1$. Die Konklusion des Lemmas gilt für Applikationsereignisse.

1.1 $\langle 2 \rangle 1$. Sei e ein Applikationsereignis. Wenn e stattfindet sendet der Prozeß eine Nachricht an alle Monitore.

Beweis: Folgt aus der Vereinbarung über auf Applikationsprozessen stattfindende Ereignisse. □

1.2 $\langle 2 \rangle 2$. Eine entsprechende Nachricht kommt nach endlicher Zeit bei allen funktionsfähigen Monitoren an und wird in Zeile 13 zugestellt.

Beweis: Folgt aus Schritt $\langle 2 \rangle 1$ und der Zuverlässigkeit der Kanäle. □

1.3 $\langle 2 \rangle 3$. Ein entsprechendes Ereignis wird in Zeile 14 an das Erkennungsmodul weitergegeben.

Beweis: Folgt aus Schritt $\langle 2 \rangle 2$ und dem Algorithmus. □

1.4 $\langle 2 \rangle 4$. Q.E.D.

Beweis: Folgt aus Schritt $\langle 2 \rangle 3$. □

2 $\langle 1 \rangle 2$. Die Konklusion des Lemmas gilt für Fehlerdetektorereignisse.

2.1 $\langle 2 \rangle 1$. Angenommen, Monitor b_j verdächtigt/rehabilitiert p_i nach e_k^i. Dann wird eine Nachricht (i, j, ac, x) in Zeile 9 verschickt.

Beweis: Folgt aus dem Algorithmus. □

2.2 $\langle 2 \rangle 2$. Eine Nachricht (i, j, ats, x) wird nach endlicher Zeit auf allen funktionsfähigen Monitoren zugestellt.

Beweis: Folgt aus Schritt $\langle 2 \rangle 1$ und der Zuverlässigkeit der Kanäle. □

2.3 $\langle 2 \rangle 3$. Q.E.D.

Beweis: In Zeilen 19/20 wird das Ereignis nur dann nicht an das Erkennungsmodul weitergegeben, wenn schon ein entsprechendes Ereignis in *fd_events* vorhanden war. In diesem Fall ist aber bereits früher ein entsprechendes Ereignis an das Deketionsmodul weitergegeben worden, da nur in Zeile 17 die Variable *fd_events* manipuliert wird. □

3 $\langle 1 \rangle 3$. Q.E.D.

Beweis: Schritte $\langle 1 \rangle 1$ und $\langle 1 \rangle 2$ decken alle Fälle ab. □

4.20 Lemma Für alle Ereignisse, die in Zeile 14 an den *possibly*-Entdeckungsalgorithmus weitergegeben werden, gibt es ein entsprechendes Ereignis in der Basisberechnung. Außerdem gilt: Die Ereignisse, die in den Zeilen 19 und 20 an das Entdeckungsmodul weitergegeben werden, sind Ereignisse gemäß der Definition des pessimistischen Fehlerdetektors, d.h. (a) es wird nur dann ein Ereignis

weitergegeben, wenn mindestens ein Monitor eine entsprechende Verdächtigung oder Rehabilitierung geäußert hat, (b) es wird maximal eine Verdächtigung pro Fehlerdetektorereignis und maximal eine Rehabilitierung gemäß Definition 4.4 erzeugt.

Beweis

1 $\langle 1 \rangle 1$. Für alle Ereignisse, die in Zeile 14 an den *possibly*-Entdeckungsalgorithmus weitergegeben werden, gibt es ein entsprechendes Ereignis in der Basisberechnung.
BEWEIS: Folgt aus dem Algorithmus und der Zuverlässigkeit der Kanäle. □

2 $\langle 1 \rangle 2$. Es gilt Teil (a) des zweiten Satzes.
BEWEIS: Folgt aus dem Algorithmus und der Zuverlässigkeit der Kanäle. □

3 $\langle 1 \rangle 3$. Es gilt Teil (b) des zweiten Satzes.
BEWEIS: Folgt aus dem Algorithmus (der Konstruktion der bedingten Anweisung). □

4 $\langle 1 \rangle 4$. Q.E.D.
BEWEIS: Folgt aus Schritten $\langle 1 \rangle 1$, $\langle 1 \rangle 2$ und $\langle 1 \rangle 3$. □

Der folgende Satz zeigt die Korrektheit des Algorithmus zur Entdeckung von *negotiably*.

4.21 Satz Sei φ ein Prädikat auf dem erweiterten Zustandsraum und habe der verwendete Algorithmus zur Entdeckung von *possibly* die im Text beschriebene Rücksetzmöglichkeit. Dann löst der Algorithmus aus Abbildung 4.10 das Entdeckungsproblem von *negotiably*(φ) für jeden Monitor, der nicht abstürzt.

Beweis

1 $\langle 1 \rangle 1$. Der Algorithmus aus Abbildung 4.10 erfüllt die Sicherheitsbedingung des Entdeckungsproblems für *negotiably*(φ) für jeden Monitor, der nicht abstürzt.
BEWEIS: Folgt aus Lemma 4.20, Lemma 4.18 und der Sicherheit des Entdeckungsalgorithmus für *possibly*. □

2 $\langle 1 \rangle 2$. Der Algorithmus aus Abbildung 4.10 erfüllt die Lebendigkeitsbedingung des Entdeckungsproblems für *negotiably*(φ) für jeden Monitor, der nicht abstürzt.
BEWEIS: Folgt aus Lemma 4.19 und den geforderten Eigenschaften (Lebendigkeit, Rücksetzmöglichkeit) des Entdeckungsalgorithmus für *possibly*. □

3 $\langle 1 \rangle 3$. Q.E.D.
BEWEIS: Folgt aus Schritten $\langle 1 \rangle 1$ und $\langle 1 \rangle 2$. □

4.6.2 Entdecken von *discernibly* bei Monitorabstürzen

Probleme mit der Definition. Im Gegensatz zu *negotiably* muß man sich bei der Betrachtung von *discernibly* nochmals mit der Definition der Beobachtungsmodalität beschäftigen, wenn auch für Monitore die *crash*-Fehlerannahme gilt. Das Prädikat *discernibly*(φ) gilt genau dann, wenn *definitely*(φ) im optimistischen Zustandsverband gilt. Im optimistischen Zustandsverband wurde ein Prozeß als abgestürzt betrachtet, wenn *alle* Monitore diesen Prozeß verdächtigten. Wenn ein Monitor aber abstürzt, kann er einen anderen Prozeß nicht mehr verdächtigen. Man muß also die Definition von *discernibly*(φ) für den Fall von Monitorabstürzen präzisieren.

4.22 Definition (*crash*-toleranter optimistischer Fehlerdetektor) Ein *crash-toleranter optimistischer Fehlerdetektor* ist ein Fehlerdetektor, der die folgenden beiden Bedingungen erfüllt:

- (Sicherheit) p_i wird genau dann nach Ereignis e_k^i verdächtigt (bzw. rehabilitiert), wenn für alle *funktionsfähigen* Beobachter b_r gilt, daß b_r den Prozeß p_i nach Ereignis e_k^i verdächtigt (bzw. rehabilitiert).
- (Lebendigkeit) Wenn für alle *funktionsfähigen* Beobachter b_r gilt, daß b_r den Prozeß p_i nach e_k^i verdächtigt (bzw. rehabilitiert), dann wird p_i nach endlicher Zeit nach e_k^i verdächtigt (bzw. rehabilitiert).

Die Definition 4.22 schließt alle abgestürzten Monitore von der Beteiligung an globalen Verdächtigungen aus.

4.23 Definition (*crash*-toleranter optimistischer Zustandsverband einer Berechnung) Ein Verband über dem erweiterten Zustandsraum einer Berechnung heißt genau dann *crash-tolerant optimistisch*, wenn die in seiner Definition verwendeten Verdächtigungs- und Rehabilitationsereignisse von einem *crash*-toleranten optimistischen Fehlerdetektor stammen.

4.24 Definition (*discernibly*(φ) bei Monitorabstürzen) Sei φ ein globales Prädikat auf dem erweiterten Zustandsraum. Wenn Monitore abstürzen dürfen, dann gilt das Prädikat *discernibly*(φ) genau dann für eine verteilte Berechnung, wenn *definitely*(φ) im *crash*-toleranten optimistischen Zustandsverband der Berechnung gilt.

Fehlalarm bei der Entdeckung von *discernibly*. Es darf mit Recht vermutet werden, daß *discernibly* in der Definition 4.24 schwierig zu entdecken sein wird. Unabhängig von der Definition kann zusätzlich noch folgendes beobachtet werden. Ein Grundproblem bei der Detektion von *discernibly* liegt darin, daß

verspätete Fehlerdetektornachrichten tatsächlich zu einer "falschen" Entdeckung führen können. Der intuitive Grund hierfür ist, daß die Entdeckung von *definitely* nicht nur von der *Menge* der konsistenten Zustände abhängt (wie bei *possibly*), sondern auch auf ihrer Anordnung zueinander.

Zur Verdeutlichung der Gefahr ist in Abbildung 4.11 ein Beispiel dargestellt, bei dem der Algorithmus vorschnell zu einer positiven Entscheidung über die Gültigkeit eines Prädikates φ kommen kann, wenn er nicht auf bestimmte Fehlerdetektornachrichten wartet. Das Beispiel besteht aus zwei Prozessen p_1 und p_2. Prozeß p_1 besitzt eine lokale Variable x und Prozeß eine lokale Variable y; beide Variablen werden nach jeweils einem Berechnungsschritt immer weiter hochgezählt. Ein Monitor b_1 beobachtet diese Berechnung und konstruiert den Verband konsistenter Zustände, der in der linken Hälfte von Abbildung 4.11 abgebildet ist. Das zu entdeckende Prädikat φ ist definiert als:

$$\varphi \equiv up[1] \wedge up[2] \wedge (x = 1 \vee y = 1)$$

Alle Zustände, in denen φ gilt, sind in der Abbildung durch ausgefüllte schwarze Kreise gekennzeichnet. Offensichtlich muß jede Beobachtung einen Zustand durchlaufen, in dem φ gilt. Demnach gilt in dieser Berechnung *definitely*(φ), und folglich auch *discernibly*(φ).

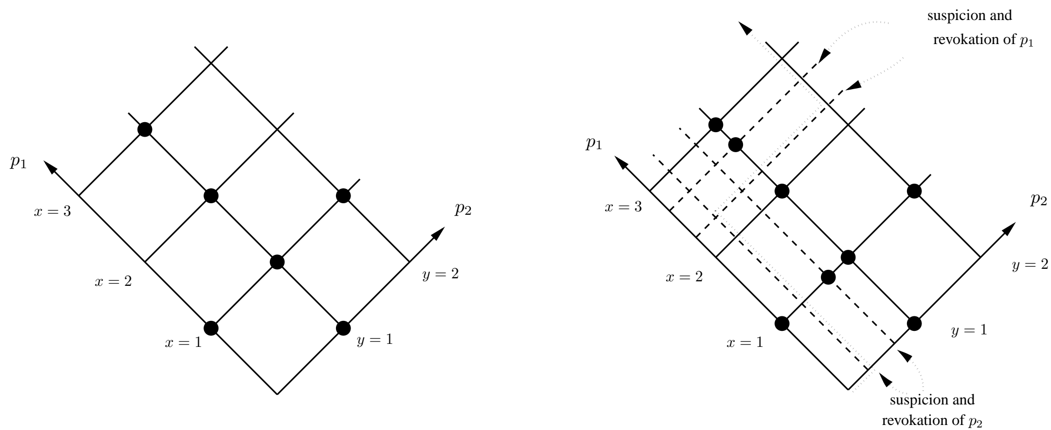


Abbildung 4.11: Beispiel, in dem fehlende Fehlerdetektornachrichten zu einer vorschnellen Entdeckung eines Prädikates φ führen können. Schwarze Punkte markieren alle Zustände, in denen φ gilt. Die gestrichelten Linien bezeichnen die Achsen, die durch verspätete Fehlerdetektornachrichten in dem Verband entstehen. Die gepunktete Linie zeigt eine Beobachtung, in der φ niemals gilt.

Nun könnte es aber sein, daß kurz nachdem der Algorithmus die Gültigkeit von *discernibly*(φ) gemeldet hat, verspätete Fehlerdetektornachrichten eintreffen, die insgesamt dazu führen, daß p_1 nach dem Ereignis $x := 2$ verdächtigt wird und p_2 vor dem Ereignis $y := 1$. Diese beiden Ereignisse müssen nachträglich in den

Zustandsverband eingebracht werden und führen zu zusätzlichen Zuständen. Der so entstehende Verband ist in der rechten Hälfte von Abbildung 4.11 dargestellt. Wieder sind alle Zustände, in denen φ gilt, durch schwarze Punkte gekennzeichnet. Nun gibt es aber plötzlich eine Beobachtung, die sich gänzlich an diesen markierten Zuständen "vorbeischlängelt" und in der somit niemals φ gilt. Demnach gilt ebenfalls nicht *definitely*(φ) und genausowenig *discernibly*(φ).

Unmöglichkeit der Entdeckung von *discernibly*. Das obige Beispiel zeigt, daß man in jedem Fall auf alle noch ausstehenden Fehlerdetektornachrichten warten muß, bis man den Zustandsverband bewertet. Da aber Monitore abstürzen können, liegt beim Warten auf derartige Nachrichten wieder das *consensus*-Dilemma vor (vergleiche Abschnitt 2.4.3): Um daraus zu entkommen, benötigt man zuverlässige Fehlerdetektoren. Da diese in dem hier gewählten Systemmodell nicht existieren, folgt, daß die Entdeckung von *discernibly* unmöglich ist.

4.25 Satz Im asynchronen Systemmodell, in dem Applikationsprozesse und Monitore abstürzen können, ist das Entdeckungsproblem für *discernibly* nicht lösbar.

Beweis

1 $\langle 1 \rangle 1$. Betrachte eine Berechnung mit einem einzigen Applikationsprozeß p_1, der von zwei Monitoren b_1 und b_2 beobachtet wird. Der Fehlerdetektor auf Monitor b_1 wird so verändert, daß b_1 Prozeß p_1 nie verdächtigt.

Beweis: Die Definition von *discernibly* ist unabhängig von den konkreten Eigenschaften des verwendeten Fehlerdetektore. Deswegen kann man einen beliebigen Fehlerdetektor annehmen. □

2 $\langle 1 \rangle 2$. Angenommen, es existiert ein Algorithmus A, der *discernibly*(φ) löst, auch wenn Monitore abstürzen. Starte A mit dem Prädikat

$$\varphi \equiv \text{“}p_1 \text{ ist abgestürzt”}$$

Wenn A meldet, daß φ gilt, dann ist b_1 abgestürzt.

Beweis: Folgt aus der Tatsache, daß b_1 den Prozeß p_1 nie verdächtigt (Schritt $\langle 1 \rangle 1$) und der Definition von *discernibly* bei Monitorabstürzen. □

3 $\langle 1 \rangle 3$. Man kann A mit dem Prädikat aus Schritt $\langle 1 \rangle 2$ verwenden, um einen Fehlerdetektor mit schwacher Genauigkeit in asynchronen Systemen zu implementieren.

Beweis: Schwache Genauigkeit bedeutet, daß ein Prozeß nie zu Unrecht verdächtigt wird. Genau dies besagt aber Schritt $\langle 1 \rangle 2$. □

4 $\langle 1 \rangle 4$. Man kann in asynchronen Systemen einen starken Fehlerdetektor implementieren.

Beweis: Verwende das Verfahren aus Schritt $\langle 1 \rangle 3$, um schwache Genauigkeit zu erreichen, sowie den Mechanismus eines unendlich häufig genauen Fehlerdetektors (Definition 4.1), um starke Vollständigkeit zu erreichen. □

5 $\langle 1 \rangle 5$. In asynchronen Systemen ist *consensus* lösbar.

Beweis: Verwende den starken Fehlerdetektor aus Schritt ⟨1⟩4 und Theorem 6.1.8 von Chandra und Toueg [35, S. 242]. □

6 ⟨1⟩6. Q.E.D.

Beweis: Schritt ⟨1⟩5 ist ein Widerspruch zum FLP-Resultat (vergleiche Kapitel 2). □

4.7 Zusammenfassung

Wer verteilte Computersysteme beherrschen möchte, der muß sie beobachten können. Beobachten bedeutet allgemein: Wissen, ob eine verteilte Berechnung eine bestimmte Eigenschaft erfüllt. Wird eine außergewöhnliche Systemeigenschaft beobachtet, die auf einen Fehler hinweist, müssen entsprechende Reaktionsmaßnahmen veranlaßt werden, um den Fehler zu tolerieren. Die Beobachtungsverfahren müssen in diesem Falle natürlich auch fehlertolerant sein.

Es gibt viele verschiedene Arten von Eigenschaften, die sich lohnen, entdeckt zu werden. Eine wichtige Klasse ist die Frage, ob ein bestimmtes globales Prädikat φ während einer Berechnung gilt oder nicht. Im fehlerfreien Fall und in asynchronen Systemen ist diese Frage nur für bestimmte Prädikatsklassen (wie stabile Prädikate) relativ einfach und eindeutig zu beantworten. Im allgemeinen Fall, d.h. wenn man die Form von φ nicht einschränkt, können unterschiedliche Beobachter zu unterschiedlichen Entscheidungen über die Gültigkeit von φ kommen. Das liegt daran, daß es keine Möglichkeit gibt, parallel auf verschiedenen Prozessen stattfindende Ereignisse in eine eindeutige "Echtzeitreihenfolge" zu bringen. Wir möchten aber gerne Fragen stellen, die von allen Beobachtern gleich beantwortet werden können. Im allgemeinen Fall muß man dann aber fragen, ob das Prädikat "möglicherweise" (d.h. könnte jemand es beobachten?) oder "definitiv" (d.h. wird jeder es beobachten?) gilt. Dies führte zu den Beobachtungsmodalitäten *possibly* und *definitely*, die in Abschnitt 4.2 vorgestellt wurden.

Die Fragestellung verkompliziert sich, wenn man asynchrone Systeme betrachtet, in denen Fehler auftreten können. Bisherige Untersuchungen auf diesem Gebiet haben die Fragestellung etwas vereinfacht, indem sie das Systemmodell verstärken und auf perfekte Fehlerdetektoren zurückgreifen [177, 184] oder die Klasse der Prädikate einschränken [70, 177, 184]. In diesem Kapitel wurde die Fragestellung erstmals "uneingeschränkt" untersucht, d.h. die Betrachtungen erfolgten im rein asynchronen Systemmodell unter der *crash*-Fehlerannahme und für allgemeine Prädikate, in denen sogar Bezug auf den funktionalen Zustand einzelner Prozesse genommen werden kann.

Unter den gewählten Voraussetzungen gibt es zunächst prinzipielle Probleme, die diskutiert werden müssen, bevor eine Lösung des eigentlichen Problemkerns

angegangen werden kann. Welche Auswirkungen beispielsweise hat der Absturz von Prozeß p_i auf die Gültigkeit eines Prädikates, das nur aus Variablen von p_i besteht? Oder wie kann man beispielsweise in asynchronen Systemen von der Gültigkeit des Prädikates "p_i ist abgestürzt" sprechen, wenn diese Frage nachweislich nicht beantwortet werden kann (vergleiche Kapitel 2)?

Die in dieser Arbeit vorgestellten Lösungen können anhand der folgenden drei Entwurfsantscheidungen charakterisiert werden:

- Rückgriff auf Ergebnisse aus der Fehlermodellierung.

  Der Absturz eines Prozesses wird behandelt, als wenn dieser eine lokale Variable *up* von *true* auf *false* setzen würde. Alle anderen Variablenwerte bleiben unverändert. Diese Idee entstammt Arbeiten auf dem Gebiet der Fehlermodellierung [16, 42, 54, 75]. Diese Entscheidung hat zwei wesentliche Vorteile: Erstens kann man Fragen nach der Gültigkeit von Prädikaten beim Absturz eines Prozesses sehr flexibel zu beantworten, indem man explizit auf den funktionalen Zustand des Prozesses innerhalb des Prädikats zurückgreift. Zweitens kann man nun auch auf relativ einfache Art und Weise ein Analogon zum Verband der konsistenten Zustände eine Berechnung definieren, ein aus der "fehlerfreien Welt" bekanntes und nützliches Konzept.

- Verbleiben im asynchronen Modell.

  Statt das reine asynchrone Modell zu verlassen und beispielsweise perfekte Fehlerdetektoren anzunehmen, zielte die Untersuchung auf Lösungen, die wenigstens potentiell in asynchronen Systemen implementiert werden können. Glücklicherweise waren die Fragestellungen unter den gegebenen Voraussetzungen größtenteils lösbar. Zunächst mußten für die gegebene Aufgabe entsprechend angepaßte Fehlerdetektoren entworfen werden, die eine zusätzliche "Kausalitätseigenschaft" besitzen (d.h. man kann eine Verdächtigung in Bezug setzen zu normalen Prozeßereignissen). Die resultierenden Detektoren sind nicht perfekt, sondern lediglich *unendlich oft genau*, d.h. während jeder Prozeßabsturz nach endlicher Zeit entdeckt wird, kann es doch zu endlich vielen fehlerhaften Verdächtigungen eines korrekten Prozesses kommen. Ein auf diesen Eigenschaften aufbauender Entdeckungsalgorithmus kann demnach nie "fehlerfrei" sein. Das ist quasi der Preis, den man für das Verbleiben im asynchronen Modell bezahlen muß.

  Daß ein Fehlerdetektor in asynchronen Systemen notwendigerweise "unsicher" sein muß, ist eine Tatsache, mit der man leben kann. Zwar folgt aus ihr, daß die Modalitäten *possibly* und *definitely* nicht mehr beobachterunabhängig sind. Jedoch kann man neue Beobachtungsmodalitäten definieren, die ihren Wahrheitswert nicht nur aus der "wahren" Gültigkeit

des zugrundeliegenden Prädikates beziehen, sondern zusätzlich aus den Informationen des Fehlerdetektors. Wenn man beispielsweise entdecken will, ob $\varphi \equiv$ "Prozeß p_i ist abgestürzt" gilt, so gibt es prinzipiell die beiden Alternativen: (1) Melde es, wenn auch nur ein lokaler Fehlerdetektor p_i verdächtigt, bzw. (2) melde es erst, wenn alle lokalen Fehlerdetektoren dies tun. In Analogie zu optimistischen und pessimistischen Netzwerkprotokollen wurde argumentiert, daß dies durchaus praktikable Konzepte sind, die die "Unsicherheit" in die Semantik neuer Modalitäten aufnehmen können. Die eingeführten Modalitäten sind demnach trotz ihrer "Unsicherheit" noch nützlich und zeigen, daß ein "Leben mit den Nachteilen" unzuverlässiger Fehlerdetektoren möglich ist.

- Modularität der Lösungen.

  Obwohl die "alten" Modalitäten *possibly* und *definitely* in asynchronen Systemen mit *crash*-Fehlern ihre Beobachterunabhängigkeit verlieren, muß man sie nicht verwerfen, da die Modalitäten *negotiably* und *discernibly* auf ihnen aufbauen. Hier kommen die verschiedenen Ideen aus den oben genannten Punkten zusammen: Der Rückgriff auf die Ergebnisse der Fehlermodellierung erlaubt es, einen Zustandsverband für den fehlerbehafteten Fall zu definieren, in dem die Informationen der lokalen Fehlerdetektoren entweder optimistisch oder pessimistisch interpretiert werden können. Auf diesen beiden alternativen Zustandsverbänden kann man nun ganz gewöhnliche *possibly*- bzw. *definitely*-Entdeckung betreiben. Entsprechend dem existentiellen Charakter von *possibly* wählt man hier die pessimistische Alternative, für *definitely* analog die optimistische. Dies hat einerseits den Vorteil, daß die neuen Modalitäten *negotiably* und *discernibly* in enger Analogie zu den altbekannten Modalitäten verstanden werden können und eine Affinität zu bestimmten Formen von Sicherheits- und Lebendigkeitseigenschaften behalten. Andererseits kann man entsprechende Entdeckungsalgorithmen unter Rückgriff auf bereits veröffentlichte Algorithmen zur Entdeckung von *possibly* und *definitely* realisieren. Dies führt zu verständlicheren, modularen Lösungen.

Theoretisch gibt es viele Anwendungsmöglichkeiten für die in diesem Kapitel vorgestellten Algorithmen. Beispielsweise kann man die Modalität *negotiably* dafür einsetzen, um Fehlerannahmen (z.B. "zu jeder Zeit sind mindestens 3 Prozesse am Leben") in der Praxis zu validieren. Fehlertoleranzverfahren reagieren in der Regel sehr sensibel auf die Verletzung der Fehlerannahme. Wenn der Entdeckungsalgorithmus also anschlägt, besteht zumindest die Möglichkeit, daß das Fehlertoleranzverfahren nicht korrekt funktioniert — eine zusätzliche Fehlerquelle.

Eine weitere Anwendungsmöglichkeit ist das *debugging* von Fehlertoleranzverfahren: Die komplexen Prädikate, die erkannt werden können (z.B. "Prozeß p_1 ist

abgestürzt und die Last auf Prozeß p_2 übersteigt Wert x") machen dies möglich. In Anlehnung der Fehlertoleranztheorie aus Kapitel 3 kann man außerdem die hier beschriebenen Entdeckungsmodalitäten auch als Detektorkomponente in beliebigen Fehlertoleranzverfahren verwenden.

Trotz der vielen Anwendungsmöglichkeiten sind die Ergebnisse dieses Kapitels leider noch weit von der Praxistauglichkeit entfernt. Dies liegt im wesentlichen daran, daß das zugrundeliegende Problem der Entdeckung von *possibly* und *definitely* allein schon sehr komplex ist [39]. In der Vergangenheit hat sich jedoch gezeigt, daß es aufbauend auf grundlegenden theoretischen Untersuchungen wie dieser durch geschickte Einschränkung der Annahmen (wie z.B. der Eigenschaften auf Prädikaten) durchaus effiziente und in der Praxis einsetzbare Verfahren entstehen können. Denn trotz der potentiellen Ästetik einer schönen Theorie sollte die praktische Verwertbarkeit nie aus den Augen verloren werden.

Kapitel 5

Zusammenfassung und Ausblick

Daß Menschenleben von korrekten Rechenergebnissen abhängen, ist keineswegs ein neues Phänomen. Schon im 19. Jahrhundert war es den Zeitgenossen bewußt, daß die Genauigkeit von menschlich berechneten Navigationstafeln für die Seefahrt über Leben und Tod der Seefahrer entschied. Nicht zuletzt diese Sorge bewog Charles Babbage, sich der Konstruktion von mechanischen und vermeintlich fehlerfreieren "Rechnern" zuzuwenden [170]. In der Nachfolge dieser Maschinen haben Computersysteme ihren Siegeszug durch nahezu alle Bereiche des täglichen Lebens angetreten. Mehr denn je hängt das gesellschaftliche Wohlergehen von der korrekten Abarbeitung von Rechenvorschriften ab. Im Gegensatz zu Softwarefehlern ist das Auftreten von Hardwarefehlern bei der Programmausführung auch theoretisch unvermeidlich. Mit Methoden der *Fehlertoleranz* kann man erreichen, daß auch im Falle derartiger Fehler das Gesamtsystem weiterhin ein definiertes (korrektes) Verhalten aufweist.

Fehlertoleranzmethoden müssen in der Praxis entworfen, implementiert und validiert werden. Je höher die Anforderungen an das System, desto schwieriger und aufwendiger werden auch die Umsetzungs- und Validierungsmaßnahmen. Dies trifft insbesondere auf *verteilte Systeme* wie das globale Internet zu. Die in heutigen Systemen erforderliche Zuverlässigkeit kann man nur erreichen, wenn man das gegebene Anwendungsgebiet genau analysiert und hinreichend formal modelliert. Trotz dieser Einsicht sind die formalen Grundlagen der Fehlertoleranz in verteilten Systemen noch nicht zufriedenstellend untersucht. Formale Grundlagen umfassen eine allgemeine Theorie und eine dadurch abgesicherte Terminologie und Methodologie. Diese Arbeit versteht sich als ein Beitrag dazu.

Kapitel 2: Modellfragen. Kapitel 2 beschäftigte sich mit Fragen der Modellierung fehlertoleranter verteilter Systeme. Nach einer Definition der relevanten Begriffe (System, Fehler, Fehlertoleranz) wurden vier prominente Systemmodelle für fehlertolerante verteilte Systeme vorgestellt und miteinander verglichen. Die

Darstellung wurde entlang der Frage entwickelt, welche Annahmen die einzelnen Modelle treffen, um das paradigmatische Fehlertoleranzproblem des *consensus* zu lösen. Die Ergebnisse dieses Kapitels können als Leitlinie dienen, um in der Praxis das "richtige" Modell für bestimmte fehlertolerante Anwendungen auszuwählen.

In der Praxis müssen die vorgestellten Modelle natürlich den jeweils gegebenen genaueren Umständen angepasst werden. Eine Vorstellung, wie dies konkret zu erreichen ist, wäre offensichtlich sehr wünschenswert, muß aber aus Platz- und Zeitgründen hier unterbleiben. Andere wichtige Modellaspekte, die auch zur Lösung von *consensus* beitragen können, konnten ebenfalls nicht berücksichtigt werden. Hierzu zählen Randomisierung [8, 31, 40] und ein neu durch Völzer vorgestelltes Fairnesskonzept [181].

Kapitel 3: Redundanz. Kapitel 3 untersuchte die nach Ansicht des Autors bisher vielversprechendste allgemeine Theorie der Fehlertoleranz, nämlich die Theorie der Detektoren und Korrektoren von Arora und Kulkarni. Die Theorie wurde in den Grundzügen vorgestellt und es wurden neue, über den aktuellen Stand der Theorie hinausgehende Ergebnisse erarbeitet. Diese Ergebnisse umfassen die vollständig formale Definition einer Begrifflichkeit der *Redundanz* und Notwendigkeitsresultate, die zeigen, welche Arten von Redundanz benötigt werden um unterschiedliche Arten von Fehlertoleranzeigenschaften zu erreichen. Diese Ergebnisse erlauben nicht nur, bestehende Fehlertoleranzmechanismen auf einer gesicherten methodologischen Basis zu analysieren, sondern sie liefern auch eine verständliche Erklärung dafür, warum und wie die Fehlertoleranztheorie von Arora und Kulkarni überhaupt "funktioniert".

Bei den Untersuchungen in Kapitel 3 ergaben sich eine Fülle weiterer Fragen, die lohnen, weiterbearbeitet zu werden:

Es ist noch offen, wie die Modellierung von Fehlern mittels der gegebenen Definition auch auf "mit dem Problem wachsenden" Fehlerannahmen ausgedehnt werden kann. Zum Beispiel wird im Bereich der sogenannten *Selbststabilisierung* [163] angenommen, daß endlich viele *beliebige* Zustandsänderungen auftreten können. Fehler wirken sich also nicht nur auf ein "originales" Programm aus, sondern auch auf alle darin (später) eingebauten Fehlertoleranzmechanismen. Auch ist noch unklar, wie sogenannte *hybride Fehlermodelle* beschrieben werden können, d.h. Fehlerannahmen der Art "es treten entweder b Byzantinische Fehler *oder* c *crash*-Fehler auf" [120]. Ein interessanter Aspekt der in dieser Arbeit verwendeten Art der Fehlermodellierung sind auch die Analogien zu existierenden Fehlerklassifikationsansätzen (wie *time/value* [150], hierarchischen Modellen [90] oder ganz allgemeinen Ansätzen [56]).

Eine Theorie ist immer angreifbar, indem die Adäquatheit der Modellierung hinterfragt wird. Beispielsweise gibt es verschiedene Alternativen zur hier gewählten

Programmsemantik, z.B. nicht-sequentielle Semantiken (*partial-order semantics*) [181] oder *branching time*-Semantiken [59]. Statt dem verwendeten Konzept der *Spurengleichheit* bei der Definition einer fehlertoleranten Version, könnte man auch *Bisimilarität* verwenden. Inwiefern sich die beschriebenen Ergebnisse verändern, bleibt abzuwarten.

Das Modell aus Kapitel 3 läßt sich auch noch verfeinern. Beispielsweise können die Lebendigkeitsannahmen wie *Maximalität* auch bezüglich einzelner Programmtransitionen definiert werden. Ein Programm muß dann nur noch "maximal" sein bezüglich einer Teilmenge von Transitionen. So könnte man einen Programmabsturz nicht als einen unendlichen *livelock* in einem bestimmten Zustand modellieren, sondern als tatsächliches "Stehenbleiben".

Wenn man die Adäquatheit der Redundanzdefinitionen untersucht, stellt sich die Frage, wie man *dynamische Redundanz* oder *strukturelle Redundanz* [55] im Rahmen des Systemmodells definieren würde. Auch ist die Redundanz, die sich als *Diversität* im Rahmen der Software-Fehlertoleranz äußert (*n-version programming*), noch nicht in das formale Rahmenwerk eingearbeitet. Offen bleibt zudem die Frage, welchen Beitrag die Ergebnisse leisten können bei der Verifikation konkreter Algorithmen [132].

Kapitel 4: Beobachtungen. Kapitel 4 wendete sich wieder konkreten algorithmischen Fragestellungen zu. Dort wurde das Erkennen von globalen Prädikaten in fehlerbehafteten verteilten Systemen untersucht. Die Literatur auf diesem Gebiet ist recht spärlich, da Beobachtungsfragen bisher nahezu ausschließlich in fehlerfreien Systemen untersucht worden sind. Kapitel 4 analysiert die dort existierenden Ansätze und überträgt die Fragestellungen, Konzepte und Algorithmen auf Systeme, in denen *crash*-Fehler auftreten können. Die gewonnenen Ergebnisse sind die ersten allgemeinen Entdeckungskonzepte für derartige Systeme.

Der in Kapitel 4 gewählte Ansatz ist sehr allgemein: Es werden keinerlei Einschränkungen auf der Menge der zu entdeckenden Prädikate gemacht. Entsprechend kann ein Resultat über die NP-Vollständigkeit des allgemeinen Beobachtungsproblems auf die gewählte Aufgabe übertragen werden. Die entworfenen Algorithmen sind demnach für den Einsatz in der Praxis ungeeignet. In Analogie zu entsprechenden Verfahren im fehlerfreien Fall kann man sicherlich die Komplexität der Aufgabe durch geschickte Einschränkung der erlaubten Prädikate stark verringern. Es bleibt offen, wie diese eingeschränkten Prädikate aussehen und wie effizient deren Entdeckungsalgorithmen wirklich sein können. Interessant wäre sicherlich auch, das Beobachtungsproblem unter schwächeren Fehlermodellen wie *crash-recovery* oder unter Zuhilfenahme von unzuverlässigen Fehlerdetektoren zu untersuchen. Hier bleibt noch ein großer Arbeitsbedarf.

Literaturverzeichnis

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.

[3] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Heartbeat: a timeout-free failure detector for quiescent reliable communication. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG97)*, pages 126–140, September 1997.

[4] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC)*, pages 231–245, September 1998.

[5] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3–30, June 1999.

[6] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash recovery model. *Distributed Computing*, 13(2):99–125, April 2000.

[7] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. On quiescent reliable communication. *SIAM Journal on Computing*, 29(6):2040–2073, December 2000.

[8] Marcos Kawazoe Aguilera and Sam Toueg. Randomization and failure detection: A hybrid approach to solve consensus. In Özalp Babaoglu and Keith Marzullo, editors, *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG96)*, volume 1151 of *Lecture Notes in Computer Science*, pages 29–39, Bologna, Italy, 9–11 October 1996. Springer-Verlag.

[9] Marcos Kawazoe Aguilera, Sam Toueg, and Borislav Deianov. On the weakest failure detector for uniform reliable broadcast. Technical Report TR99-1741, Cornell University, Computer Science, April 30, 1999.

[10] Carlos Almeida and Paulo Veríssimo. Timing failure detection and real-time group communication in quasi-synchronous systems. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, L'Aquila, Italy, June 1996.

[11] Carlos Almeida, Paulo Veríssimo, and António Casimiro. The quasi-synchronous approach to fault-tolerant and real-time communication and processing. Technical Report CTI RT-98-04, Instituto Superior Técnico, Lisboa, Portugal, July 1998.

[12] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[13] Anish Arora and Mohamed Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[14] Anish Arora and Sandeep S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

[15] Anish Arora and Sandeep S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)*, May 1998.

[16] Anish Kumar Arora. *A foundation of fault-tolerant computing*. PhD thesis, The University of Texas at Austin, December 1992.

[17] Rada Krishan Arora and M. N. Gupta. More comments on "Distributed termination detection algorithm for distributed computations" (Letter to the Editor). *Information Processing Letters*, 29:53–55, September 1988.

[18] Rada Krishan Arora, S. P. Rana, and M. N. Gupta. Distributed termination detection algorithm for distributed computations. *Information Processing Letters*, 22:311–314, May 1986.

[19] Algirdas Avižienis. Fault-tolerant systems. *IEEE Transactions on Computers*, 25(12):1304–1312, December 1976.

[20] Özalp Babaoğlu, Alberto Bartoli, and Gianluca Dini. Replicated file management in large-scale distributed systems. In *WDAG94 Distributed Algorithms 8th International Workshop Proceedings, Springer-Verlag LNCS:857*, pages 1–16, 1994.

[21] Özalp Babaoğlu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In Mullender [140], chapter 4, pages 55–96.

[22] Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG96)*, pages 105–122, 1996.

[23] Joffroy Beauquier, Sylvie Delaët, Shlomi Dolev, and Sébastien Tixeuil. Transient fault detectors. In *Proceedings of the 12th International Symposium on DIStributed Computing (DISC'98)*, number 1499 in Lecture Notes in Computer Science, pages 62–74, Andros, Greece, September 1998. Springer-Verlag.

[24] Joffroy Beauquier and Synnöve Kekkonen-Moneta. Fault-tolerance and self-stabilization: impossibility results and solutions using self-stabilizing failure detectors. *International Journal of System Science*, 28(11):1177–1187, 1997.

[25] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proc. Second Ann. ACM Symp. on Principles of Distributed Computing*, pages 27–30, 1983.

[26] Ute Bernhardt. Reiten auf der Risikowelle (Editorial zum Sonderheft zum Thema "Verletzlichkeit der Informationsgesellschaft"). *FIfF Kommunikation*, (3):3, September 2000.

[27] Romain Boichat and Rachid Guerraoui. Reliable broadcast in the crash-recovery model. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)*, Nürnberg, Germany, October 2000. IEEE Computer Society Press.

[28] Malte Borcherding. *Authentifikationsvoraussetzungen für effiziente byzantinische Übereinstimmung*. PhD thesis, Universität Karlsruhe, Fakultät für Informatik, 1997. Logos-Verlag, Berlin.

[29] Jonathan Bowen and Victoria Stravridou. Safety-critical systems, formal methods and standards. *IEE/BCS Software Engineering Journal*, 8(4):189–209, July 1993.

[30] Stanley N. Burris and H. P. Sankappanavar. *A course in universal algebra*. Springer-Verlag, 1981. Revised edition online at `http://thoralf.uwaterloo.ca/htdocs/ualg.html`.

[31] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 123–132, Portland, Oregon, 2000.

[32] S. Chandra and P.M. Chen. How fail-stop are faulty programs? In *Proceedings of the 28th IEEE Symposium on Fault Tolerant Computing Systems (FTCS-28)*, pages 240–249, June 1998.

[33] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[34] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing (PODC'91)*, pages 325–340, 1991.

[35] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[36] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, Mass., 1988.

[37] Bernadette Charron-Bost, Carole Delporte-Gallet, and Hugues Fauconnier. Local and temporal predicates in distributed systems. *ACM Transactions on Programming Languages and Systems*, 17(1):157–179, January 1995.

[38] Bernadette Charron-Bost, Sam Toueg, and Anindya Basu. Revisiting safety and liveness in the context of failures. In C. Palamidessi, editor, *Proceedings of CONCUR2000 – Concurrency Theory, 11th Int. Conference*, number 1877 in Lecture Notes in Computer Science, pages 552–565, University Park, PA, August 2000. Springer-Verlag.

[39] Craig M. Chase and Vijay K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[40] Benny Chor and Cynthia Dwork. Randomization in Byzantine agreement. In *Advances in Computing Research 5: Randomness and Computation*, pages 443–497. JAI Press, 1989.

[41] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. *ACM SIGPLAN Notices*, 26(12):167–174, December 1991.

[42] Flaviu Cristian. A rigorous approach to fault-tolerant programming. *IEEE Transactions on Software Engineering*, 11(1):23–31, January 1985.

[43] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[44] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[45] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6), June 1999.

[46] Edsger W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[47] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[48] Edsger W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219, June 1983.

[49] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[50] Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. Failure detectors in omission failure environments. Technical Report TR96-1608, Cornell University, Computer Science Department, September 1996.

[51] Danny Dolev, Roy Friedmann, Idit Keidar, and Dahlia Malkhi. Failure detectors in omission failure environments. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC97)*, 1997.

[52] Assai Doudou and André Schiper. Muteness detectors for consensus with Byzantine processes. Technical report, EPFL – Département d'Informatique, Lausanne, Switzerland, 1997.

[53] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[54] Klaus Echtle. Fehlermodellierung bei Simulation und Verifikation von Fehlertoleranz-Algorithmen für Verteilte Systeme. In F. Belli, S. Pfleger, and M. Seifert, editors, *Software-Fehlertoleranz und -Zuverlässigkeit*, number 83 in Informatik-Fachberichte, pages 73–88. Springer-Verlag, 1984.

[55] Klaus Echtle. *Fehlertoleranzverfahren*. Springer-Verlag, 1990.

[56] Klaus Echtle and Asif Masum. Understanding cooperative byzantine failures: A novel failure classification to enable efficient fault-tolerant protocols. In *Proceedings of the Annual IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems (FTPDS'99)*, San Juan, Puerto Rico, USA, April 1999. Kluwer.

[57] Klaus Echtle and Asif Masum. A fundamental failure model for fault-tolerant protocols. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium (IPDS2K)*, pages 69–78, Chicago, IL, 2000. IEEE Computer Society Press.

[58] Klaus Echtle and João Gabriel Silva. Fehlerinjektion – ein Mittel zur Bewertung der Maßnahmen gegen Fehler in komplexen Rechnersystemen. *Informatik Spektrum*, 21(6):328–336, December 1998.

[59] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 997–1072. Elsevier, 1990.

[60] Didier Essame, Jean Arlat, and David Powell. Padre: A protocol for asymmetric duplex redundancy. In *Proceedings of the Seventh IFIP International Working Conference on Dependable Computing for Critical Applications*, San Jose, USA, January 1999.

[61] Pascal Felber, Xavier Défago, Rachid Guerraoui, and P. Oser. Failure detectors as first class objects. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99)*, pages 132–141, Edinburgh, Scotland, September 1999.

[62] Pascal Felber, Rachid Guerraoui, and Mohamed E. Fayad. Putting OO distributed programming to work. *Communications of the ACM*, 42(11):97–101, November 1999.

[63] Christof Fetzer. A comparison of timed asynchronous systems and asynchronous systems with failure detectors. In *Proceedings of the Third European Research Seminar on Advances in Distributed Systems (ERSADS'99)*, pages 109–118, Madeira Island, Portugal, April 1999.

[64] Christof Fetzer and Flaviu Cristian. On the possibility of consensus in asynchronous systems. In *Proceedings of the 1995 Pacific Rim International Symposium on Fault-Tolerant Systems*, December 1995.

[65] Christof Fetzer and Flaviu Cristian. Fail-aware failure detectors. In *Proceedings of the 15th Symposium on Reliable Distributed Systems (SRDS 1996)*, pages 200–209, Los Alamitos, Ca., USA, October 1996. IEEE Computer Society Press.

[66] Christof Fetzer and Flaviu Cristian. Fail-awareness: An approach to construct fail-safe applications. In *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 282–291. IEEE, June 1997.

[67] Colin J. Fidge. Timestamps in message-passing systems that preserve partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, February 1988.

[68] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[69] Nissim Francez. *Fairness.* Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[70] Vijay K. Garg and J. Roger Mitchell. Distributed predicate detection in a faulty environment. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)*, 1998.

[71] Vijay K. Garg and J. Roger Mitchell. Implementable failure detectors in asynchronous systems. In *Proc. 18th Conference on Foundations of Software Technology and Theoretical Computer Science*, number 1530 in Lecture Notes in Computer Science, Chennai, India, December 1998. Springer-Verlag.

[72] Felix C. Gärtner. Specifications for fault tolerance: A comedy of failures. Technical Report TUD-BS-1998-03, Darmstadt University of Technology, Darmstadt, Germany, October 1998.

[73] Felix C. Gärtner. An exercise in systematically deriving fault-tolerance specifications. In *Proceedings of the Third European Research Seminar on Advances in Distributed Systems (ERSADS)*, Madeira Island, Portugal, April 1999.

[74] Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, March 1999.

[75] Felix C. Gärtner. Transformational approaches to the specification and verification of fault-tolerant systems: Formal background and classification. *Journal of Universal Computer Science (J.UCS)*, 5(10):668–692, October 1999. Special Issue on Dependability Evaluation and Assessment.

[76] Felix C. Gärtner and Sven Kloppenburg. Consistent detection of global predicates under a weak fault assumption. In *Proceedings of the 19th IEEE*

Symposium on Reliable Distributed Systems (SRDS2000), pages 94–103, Nürnberg, Germany, October 2000. IEEE Computer Society Press.

[77] Felix C. Gärtner and Henning Pagnia. Enhancing the fault tolerance of replication: another excercise in constrained convergence. In *Digest of FastAbstracts of the 28th IEEE Symposium on Fault Tolerant Computing Systems (FTCS-28)*, pages 29–30, June 1998.

[78] Felix C. Gärtner and Henning Pagnia. Self-stabilizing load distribution for replicated servers on a per-access basis. In Anish Arora, editor, *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilizing Systems*, pages 102–109, Austin, TX, June 1999. IEEE Computer Society Press.

[79] Felix C. Gärtner, Henning Pagnia, and Holger Vogt. Approaching a formal definition of fairness in electronic commerce. In *Proceedings of the International Workshop on Electronic Commerce (WELCOM'99)*, pages 354–359, Lausanne, Switzerland, October 1999. IEEE Computer Society Press.

[80] Felix C. Gärtner and Hagen Völzer. Redundancy in space in fault-tolerant systems. Technical Report TUD-BS-2000-06, Department of Computer Science, Darmstadt University of Technology, Darmstadt, Germany, July 2000.

[81] Richard A. Golding. Accessing replicated data in a large-scale distributed system. Master's thesis, University of California, Santa Cruz, June 1991. Also published as Technical Report UCSC-CRL-91-18.

[82] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[83] Rachid Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In Jean-Michel Hélary and Michel Raynal, editors, *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG95)*, volume 972 of *Lecture Notes in Computer Science*, pages 87–100, Le Mont-Saint-Michel, France, 13–15 September 1995. Springer-Verlag.

[84] Rachid Guerraoui, Mikel Larrea, and André Schiper. Non blocking atomic commitment with an unreliable failure detector. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems (SRDS95)*, September 1995.

[85] Rachid Guerraoui and André Schiper. "Gamma-accurate" failure detectors. In Özalp Babaoglu and Keith Marzullo, editors, *Distributed Algorithms, 10th International Workshop, WDAG '96*, volume 1151 of *Lecture Notes*

in Computer Science, pages 269–286, Bologna, Italy, 9–11 October 1996. Springer-Verlag.

[86] Rachid Guerraoui and André Schiper. Consensus: the big misunderstanding. In *Proceedings of the 6th Workshop on Future Trends of Distributed Computing Systems (FTDCS-6)*, October 1997.

[87] Rachid Guerraoui and André Schiper. Genuine atomic multicast. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG97)*, number 1320 in Lecture Notes in Computer Science, pages 141–154. Springer-Verlag, September 1997.

[88] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, April 1997.

[89] Vassos Hadzilacos. *Issues of Fault Tolerance in Concurrent Computations.* PhD thesis, Harvard University, 1984. also published as Technical Report TR11-84.

[90] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Mullender [140], chapter 5, pages 97–145.

[91] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Computer Science Department, May 1994.

[92] Joseph Y. Halpern and Aleta Ricciardi. A knowledge-theoretic analysis of uniform distributed coordination and failure detectors. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC'99)*, pages 73–82, 1999.

[93] Walter L. Heimerdinger and Chuck B. Weinstock. A conceptual framework for system fault tolerance. Technical Report CMU/SEI-92-TR-33, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, October 1992.

[94] Maurice P. Herlihy and Jeannette M. Wing. Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93–104, January 1991.

[95] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, April 1997.

[96] Michel Hurfin, A. Mostéfaoui, and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98)*, pages 280–286, West Lafayette, Indiana, October 1998. IEEE Computer Society Press.

[97] Reinhard Hutter. Angriffe auf Informationstechnik und Infrastrukturen – Realität oder Science Fiction? *Aus Politik und Zeitgeschichte*, 41–42:31–38, 2000.

[98] Pankaj Jalote. *Fault tolerance in distributed systems*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1994.

[99] Roland Jégou, Raoul Medina, and Lhouari Nourine. Linear space algorithm for on-line detection of global predicates. In Jörg Desel, editor, *Proceedings of the International Workshop on Structures in Concurrency Theory (STRICT)*, Workshops in Computing, pages 175–189. Springer-Verlag, 1995.

[100] Anita Jones. The challenge of building survivable information-intensive systems (introduction to special issue on "critical infrastructures"). *IEEE Computer*, 33(8):39–43, August 2000.

[101] Synnöve Kekkonen. *Résistance aux Fautes dans les Algorithmes Répartis: Auto-Stabilisation et Tolérance aux Fautes*. PhD thesis, Université de Paris-Sud, France, 1998.

[102] Sven Kloppenburg. Entdecken globaler Prädikate in verteilten systemen mit Anhalteausfällen. Diplomarbeit, Technische Universität Darmstadt, Fachbereich Informatik, Fachgebiet Betriebssysteme, September 1999. DA-BS-1999-02.

[103] Sven Kloppenburg and Felix C. Gärtner. Consistent detection of global predicates in asynchronous systems with crash failures. Technical Report TUD-BS-2000-01, Darmstadt University of Technology, Department of Computer Science, Darmstadt, Germany, February 2000.

[104] Hermann Kopetz and Paulo Veríssimo. Real time and dependability concepts. In Mullender [140], chapter 16, pages 411–446.

[105] Christoph Kreitz, Mark Hayden, and Jason Hickey. A proof environment for the development of group communication systems. In H. Kirchner, editor, *15th International Conference on Automated Deduction*, Lecture Notes in AI. Springer-Verlag, 1998.

[106] D. Richard Kuhn. Sources of failure in the public switched telephone network. *IEEE Computer*, 30(4):31–36, April 1997.

[107] Sandeep S. Kulkarni. *Component Based Design of Fault-Tolerance*. PhD thesis, Department of Computer and Information Science, The Ohio State University, 1999.

[108] Sandeep S. Kulkarni and Anish Arora. Automating the addition of fault-tolerance. In Mathai Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium (FTRTFT 2000) Proceedings*, number 1926 in Lecture Notes in Computer Science, pages 82–93, Pune, India, September 2000. Springer-Verlag.

[109] Peter Ladkin. Re: A340 incident at Heathrow (Hatton, RISKS-16.92). The Risks Digest (Forum on Risks to the Public in Computers and Related Systems), March 1995.

[110] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[111] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[112] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[113] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[114] Leslie Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, August/September 1995.

[115] Leslie Lamport and Nancy Lynch. Distributed computing: models and methods. In *Handbook of Theoretical Computer Science (Volume B: Formal Models and Semantics)*, chapter 18, pages 1157–1199. Elsevier, 1990. J. van Leeuwen, Editor.

[116] Jean-Claude Laprie, editor. *Dependability: Basic concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, 1992.

[117] Mikel Larrea, Antonio Fernández, and Sergio Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)*, Nürnberg, Germany, October 2000. IEEE Computer Society Press.

[118] Gerard Le Lann. On real-time and non real-time distributed computing. In *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG95)*, pages 51–70, September 1995.

[119] Peter A. Lee and Thomas Anderson. *Fault Tolerance: Principles and Practice*. Dependable computing and fault-tolerant systems. Springer-Verlag, Berlin ; New York, second edition, 1990.

[120] Patrick Lincoln and John Rushby. Formal verification of an algorithm for interactive consistency under a hybrid fault model. In Costas Courcoubetis, editor, *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, pages 292–304, Elounda, Greece, June/July 1993. Springer-Verlag.

[121] Barbara Liskov and William Weihl. Specifications of distributed programs. *Distributed Computing*, 1:102–118, 1986.

[122] Zhiming Liu and Mathai Joseph. Transformation of programs for fault-tolerance. *Formal Aspects of Computing*, 4(5):442–469, 1992.

[123] Zhiming Liu and Mathai Joseph. Specification and verification of recovery in asynchronous communicating systems. In Jan Vytopil, editor, *Formal Techniques in Real-time and Fault-tolerant Systems*, chapter 6, pages 137–165. Kluwer, 1993.

[124] Wai-Kau Lo and Vassos Hadzilacos. Using failure detectors to solve consensus in asynchronous shared-memory systems (extended abstract). In Gerard Tel and Paul M. B. Vitányi, editors, *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG94)*, volume 857 of *Lecture Notes in Computer Science*, pages 280–295, Terschelling, The Netherlands, 29 September–1 October 1994. Springer-Verlag.

[125] Darrel D. E. Long, John L. Carroll, and C. J. Park. A study of the reliability of Internet sites. In *Proceedings of the 10th IEEE Symposium on Reliable Distributed Systems (SRDS91)*, pages 177–186, September 1991.

[126] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[127] Nancy A. Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.

[128] Maftia home – Malicious- and Accidental-Fault Tolerance for Internet Applications. Internet: `http://www.newcastle.research.ec.org/maftia/`.

[129] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer-Verlag, 1991.

[130] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag, 1995.

[131] Heiko Mantel. Possibilistic definitions of security - an assembly kit. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop,*, Cambridge, England, July 2000. IEEE Computer Society Press.

[132] Heiko Mantel and Felix C. Gärtner. A case study in the mechanical verification of fault tolerance. *Journal of Experimental & Theoretical Artificial Intelligence (JETAI)*, 12(4):473–488, October 2000.

[133] Keith Marzullo and Gil Neiger. Detection of global state predicates. In *Proceedings of the 5th International Workshop on Distributed Algorithms (WDAG91)*, pages 254–272, 1991.

[134] Asif Masum. *Non-cooperative Byzantine failures: A new framework for the design of efficient fault tolerance protocols*. PhD thesis, Universität-Gesamthochschule Essen, Fachbereich Mathematik und Informatik, 2000. Published by Libri Books on demand, ISBN 3-8311-0815-3.

[135] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, 1987.

[136] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Chateau de Bonas, France, 1989. Elsevier Science Publishers. Reprinted on pages 123–133 in [183].

[137] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, pages 79–93, Oakland, CA, 1994.

[138] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[139] David L. Mills. Network time protocol (version 3). Internet Request for Comments RFC 1305, March 1992.

[140] Sape Mullender, editor. *Distributed Systems*. Addison-Wesley, second edition, 1993.

[141] Peter G. Neumann. *Computer Related Risks*. ACM Press, 1995.

[142] Jens Nordahl. Design for dependability. In Carl E. Landwehr, editor, *Proceedings of the third IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-3)*, pages 29–38. Springer-Verlag, 1993.

[143] Rui Oliveira, Rachid Guerraoui, and André Schiper. Consensus in the crash-recover model. Technical Report TR-97/239, EPFL – Départment d'Informatique, Lausanne, Switzerland, August 1997.

[144] Henning Pagnia and Holger Vogt. Exchanging goods and payment in electronic business transactions. In *Proceedings of the Third European Research Seminar on Advances in Distributed Systems (ERSADS)*, Madeira Island, Portugal, April 1999.

[145] Henning Pagnia, Holger Vogt, Felix C. Gärtner, and Uwe G. Wilhelm. Solving fair exchange with mobile agents. In *Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents (ASA/MA2000)*, volume 1882 of *Lecture Notes in Computer Science*, pages 57–72, Zurich, Switzerland, September 2000. Springer-Verlag.

[146] Doron Peled and Mathai Joseph. A compositional framework for fault-tolerance by specification transformation. *Theoretical Computer Science*, 128:99–125, 1994.

[147] Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, 12(3):477–482, March 1986.

[148] Stefan Pleisch and André Schiper. Modeling fault-tolerant mobile agent execution as a sequence of agreement problems. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)*, pages 11–20, Nürnberg, Germany, October 2000. IEEE Computer Society Press.

[149] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, October 31–November 2 1977. IEEE, IEEE Computer Society Press.

[150] David Powell. Failure mode assumptions and assumption coverage. In Dhiraj K. Pradhan, editor, *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing (FTCS '92)*, pages 386–395, Boston, MA, July 1992. IEEE Computer Society Press.

[151] I. S. W. B. Prasetya and S. D. Swierstra. Factorizing fault tolerance. Technical Report UU-CS-2000-02, University of Utrecht, Department of Computer Science, Utrecht, The Netherlands, 2000. Appears in special issue of TCS on fault tolerance.

[152] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[153] T. R. N. Rao and E. Fujiwara. *Error-control coding for computer systems*. Prentice-Hall, 1989.

[154] Kurt Rothermel and Markus Straßer. A Fault-Tolerant Protocol for Providing the Exactly-Once Property of Mobile Agents. In *Proc. 17th IEEE Symposium on Reliable Distributed Systems 1998 (SRDS'98)*, pages 100–108, Los Alamitos, California, 1998. IEEE Computer Society Press.

[155] John Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and System Safety*, 43(2):189–219, 1994.

[156] Laura S. Sabel and Keith Marzullo. Election vs. consensus in asynchronous systems. Technical Report TR95-1488, Cornell University, Computer Science Department, February 1995.

[157] Beverly Sanders. A predicate transformer approach to knowledge and knowledge-based protocols. In Luigi Logrippo, editor, *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing (PODC'91)*, pages 217–22, Montéal, Québec, Canada, August 1991. ACM Press.

[158] Henk Schepers. Tracing fault tolerance. In Carl E. Landwehr, editor, *Proceedings of the third IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-3)*, pages 39–48. Springer-Verlag, 1993.

[159] André Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[160] Richard D. Schlichting and Fred B. Schneider. Fail stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[161] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[162] Fred B. Schneider. What good are models and what models are good? In Sape Mullender, editor, *Distributed Systems*, chapter 2, pages 17–26. Addison-Wesley, second edition, 1993.

[163] Marco Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, 1993.

[164] Christiane Schulzki-Haddouti. Kritische Infrastrukturen. *FIfF Kommunikation*, pages 19–20, September 2000.

[165] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing*, 7:149–174, 1994.

[166] Nicole Sergent, Xavier Défago, and André Schiper. Failure detectors: implementation issues and impact on consensus performance. Technical Report SSC/1999/019, École Polytechnique Fédérale de Lausanne, Switzerland, 1999.

[167] J. T. Sims. Redundancy management software services for seawolf ship control system. In *Proceedings of the 27th IEEE Symposium on Fault Tolerant Computing Systems (FTCS-27)*, pages 390–394, Washington - Brussels - Tokyo, June 1997. IEEE.

[168] Scott D. Stoller and Fred B. Schneider. Faster possibility detection by combining two approaches. In *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG95)*, pages 318–332, September 1995.

[169] Scott D. Stoller, Leena Unnikrishnan, and Yanhong A. Liu. Efficient detection of global properties in distributed systems using partial-order methods. In *Computer Aided Verification (CAV 2000)*, volume 1855 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[170] D. Swade. The construction of Charles Babbage's difference engine. *Annals of the History of Computing.*, 13(1):82–83, 1991.

[171] Richard B. Tan, Gerard Tel, and Jan van Leeuwen. Comments on "Distributed termination detection algorithm for distributed computations" (Letter to the Editor). *Information Processing Letters*, 23:163, October 1986.

[172] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[173] Oliver Theel and Felix C. Gärtner. On proving the stability of distributed algorithms: self-stabilization vs. control theory. In Vladimir B. Bajic, editor, *Proceedings of the International Systems, Signals, Control, Computers Conference (SSCC'98), Durban, South Africa*, volume III, pages 58–66, September 1998.

[174] Oliver Theel and Felix C. Gärtner. An exercise in proving convergence through transfer functions. In Anish Arora, editor, *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilizing Systems*, pages 41–47, Austin, TX, June 1999. IEEE Computer Society Press.

[175] John Turek and Dennis Shasha. The many faces of consensus in distributed systems. *IEEE Computer*, 25(6):8–17, June 1992.

[176] A. J. M. van Gasteren and Gerard Tel. Comments on "on the proof of a distributed algorithm": always true is not invariant. *Information Processing Letters*, 35:277–279, September 1990.

[177] Subbarayan Venkatesan. Reliable protocols for distributed termination detection. *IEEE Transactions on Reliability*, 38(1):103–110, April 1989.

[178] Paulo Veríssimo. Real-time communication. In Mullender [140], chapter 17, pages 447–490.

[179] Holger Vogt and Henning Pagnia. Fairer Austausch beim elektronischen Einkauf im Internet. In *Proceedings of the 6th DFN-CERT Workshop "Sicherheit in vernetzten Systemen"*, Hamburg, Germany, March 1999.

[180] Holger Vogt, Henning Pagnia, and Felix C. Gärtner. Modular fair exchange protocols for electronic commerce. In *Proceedings of the 15th Annual Computer Security Applications Conference*, pages 3–11, Phoenix, Arizona, December 1999. IEEE Computer Society Press.

[181] Hagen Völzer. *Fairness, Randomisierung und Konspiration in verteilten Algorithmen.* PhD thesis, Humboldt Universität zu Berlin, Fakultät für Informatik, December 2000.

[182] D. G. Weber. Formal specification of fault-tolerance ad its relation to computer security. In Sol Greenspan, editor, *Proceedings of the 5th International Workshop on Software Specification and Design*, pages 273–277, Pittsburgh, PA, May 1989. IEEE Computer Society Press.

[183] Zhonghua Yang and T. Anthony Marsland, editors. *Global States and Time in Distributed Systems.* IEEE Computer Society Press, 1994.

[184] Pei yu Li and Bruce McMillin. Fault-tolerant distributed deadlock detection/resolution. In *Proceedings of the 17th Annual International Computer Software and Applications Conference (COMPSAC'93)*, pages 224–230, November 1993.

Index

Δ, 41
$\cdot$ (Konkatenation), 67
δ, 41
$\perp$, 120
$\Box$ (“immer”), 71
$\Box$ (“immer”), 28
$\Diamond$ (“nach endlicher Zeit”), 28, 71
$\rightarrow$ (Kausalitätsrelation), 114
$\sigma \in \Sigma$, 70
$\sigma \in prop(\Sigma)$, 70
$\sigma|_i$, 67

Abarbeitungsgeschwindigkeiten, 41
Ablauf, 26
Ablauf über C, 67
Ablaufpräfix, 67
ac, 124
agreement, 38
agreement problems, 36
Aguilera, M.K., 59
aktive Wache, 64
Almeida, C., 3, 5, 57
Alpern, B., 30, 68, 72, 105
Anhalteausfälle, 34
Anweisung, 64
Anzeigeprädikat, 64
application process, 113
application message, 113
Applikationsnachricht, 113
Applikationsprozeß, 113
Applikationsuhr, 132
Arora, A., 3, 5, 19, 61–63, 67, 70, 72, 78, 79, 84, 107, 158
ARQ, 101
assertional reasoning, 29
asynchrones Systemmodell, 31
ATM, 58
atomic broadcast, 39
atomic commitment, 39
Aushungerungsfreiheit, 29
automatic repeat request, 101
availability, 17

Babbage, C., 157
Basisberechnung, 113
bedingte Lebendigkeitseigenschaft, 55
Beobachter, 113
Beobachterprozeß, 113
Beobachterunabhängigkeit, 114
Beobachtung, 116
Beobachtungsmodalität, 112
Beobachtungsmodalitat, 117
Beobachtungsproblem, 111, 114
Betriebssysteme, 28
bewachte Anweisungen, 63
Bewahrung einer Spezifikation (Definition), 73
Bisimilarität, 159
bottom-Element, 116
branching time, 159
broadcast, 31, 108
Byzantine, 34
byzantinisches Fehlermodell, 34

C (Menge von Zuständen), 67
Casimiro, A., 3, 5, 57
Chandra, T.D., 3, 5, 48, 51, 52, 122, 153
Chase, C.M., 118
Chen, W., 60
command, 64
common, 135

conditional timeliness property, 55
consensus, 38, 158
consensus-Problem, 38
control message, 113
Cooper, R., 3, 5
crash, 34
crash-recovery, 34, 53, 60, 159
crash-toleranter optimistischer Fehlerdetektor (Definition), 150
crash-toleranter optimistischer Zustandsverband einer Berechnung (Definition), 150
Cristian, F., 3, 5, 55
cut, 115

decide, 38
definitely(φ), 117
dependability, 17
design, 79
detectors and correctors, 62
Detektor, 64
Detektoren und Korrektoren, 62
discernibly(φ) (Definition), 129
discernibly(φ) bei Monitorabstürzen (Definition), 150
Diversität, 159
Dolev, D., 42
Dwork, C., 3, 5, 42
dynamische Redundanz, 159

Eigenschaft, 27
Eigenschaft über C, 67
e_i^k, 114
einfaches Programm, 68
election, 39
Endzustände, 68
enlargening behavior property, 75
Ereignis, 114
Erfüllung einer Eigenschaft durch einen Ablauf, 67
Erfüllung einer Spezifikation durch ein Programm, 73
Erkennen von Prädikaten, 48
erweiterter Zustandsraum, 121
Erweiterung eines Programmes (Definition), 81
event, 114
eventual weak accuracy, 50
eventually weak failure detector, 51
exactly once copy, 37
execution, 26
extended state space, 121

$F(\Sigma)$, 75
$F(A)$, 75
$F(T)$, 75
fail-awareness, 54
fail-safe-Fehlertoleranz, 78
fail-silent, 34
fail-softness, 78
fail-stop, 34
failure detector modules, 48
failure detectors, 47
failure mode, 78
failure semantics, 78
faire Konfliktauflösung, 71
fairer Austausch, 37
Fairneßannahme, 64, 68
fault assumption, 33, 78
fault coverage, 36
fault model, 33
fault-affected behavior, 78
Fehlerdetektormodule, 48
Fehlerannahme, 33, 75
fehlerbehaftete Version, 75
Fehlerbereich, 35
Fehlerbewußtsein, 54
fehlerbezogene Spezifikation, 78
Fehlerdetektoren, 47
Fehlerdiagnose, 37
Fehlererfassung, 36
fehlerhafte Prozesse, 35
Fehlerinjektion, 34
Fehlermodell, 33
Fehlermodell (Definition), 74

Fehlermodelle sind verhaltenserweiternd (Proposition), 75
fehlertolerante Version (Definition), 81
Fehlertoleranz, 18, 157
Fetzer, C., 3, 5, 55, 59
Fischer, M., 39
flooding, 108
FLP, 39
Fortschrittsannahme, 55, 68
fusion closed properties, 72
fusions-abgeschlossene Eigenschaften, 72
Fusions-Abgeschlossenheit (Definition), 72
Fusionsabgeschlossenheit, 84

Gültigkeit, 38
Garg, V.K., 118, 124
general omission, 34
global state, 26, 115
globale Fehlerannahme, 35
globaler Zustand, 26, 115
Gouda, M.G., 61, 78, 79
graceful degradation, 78
GSM, 58
guard, 64
guarded commands, 63

Hadzilacos, V., 35, 52
Hamming-Kodes, 107
Hinzufugung einer Transition (Definition), 86
history variables, 72
hybride Fehlermodelle, 158

$I \cdot C^*$, 76
inf, 116
infinitely often accurate, 122
inkonsistente Beobachtung, 116
inkonsistenter Zustand, 115
Integrität, 17
integrity, 17
intrusionstolerant, 108
Invariante, 27, 29
ISDN, 58

Joseph, M., 79, 82

kausal-konsistente Beobachtung, 116
kausaler *broadcast*, 116
kausaler unendlich häufig genauer Fehlerdetektor (Definition), 123
Kausalitätsrelation, 114
keine Fehlertoleranz ohne Redundanz (Theorem), 105
Kodierungstheorie, 106
Kommunikationskanäle, 25
Kommunikationssubsystem, 26
Konflikt, 70
Konkatenation, 67
konsistente Beobachtung, 116
konsistenter Zustand, 115
Kontrollnachricht, 113
Koordinator, 43
korrekte Prozesse, 35
Korrektor, 65
kritische Computersysteme, 17
kritische Infrastrukturen, 17
Kryptographie, 107
Kulkarni, S.S., 3, 5, 19, 62, 63, 67, 70, 72, 78, 84, 107, 158

Lamport, L., 20
Laprie, J.-C., 61
lattice, 116
Lebendigkeitsannahme, 68
Lebendigkeitsannahme (Definition), 69
Lebendigkeitseigenschaft, 29
Lebendigkeitseigenschaft (Definition), 68
lebendigkeitskonservatives Fehlermodell (Definition), 93
Lebendigkeitsspezifikation, 73
letztendlich schwache Genauigkeit, 50
letztendlich schwacher Fehlerdetektor, 51
Li, P., 118

Liu, Z., 79
livelock, 159
liveness property, 29
local algorithm, 26
local execution, 26
local predicate, 120
local state, 26, 115
lokale Berechnung, 26
lokale Fehlerannahme, 35
lokale Fehlerdetektoren, 48
lokale Zustand, 26
lokaler Algorithmus, 26
lokaler Zustand, 115
lokales Prädikat, 120
Lynch, N., 3, 5, 39

m (Anzahl der Beobachtungsprozesse), 113
maintains, 73
Marzullo, K., 3, 5
maskierende Fehlertoleranz, 78
masking fault-tolerance, 78
Maximalität, 69, 159
McMillin, B., 118
Mengenredunktionsprädikate, 118
minimal verletzender Präfix (Definition), 87
Mitchell, J.R., 118, 124
mobile Agenten, 37
monitor process, 113
Monitorprozeß, 113
Monitoruhr, 133
Monotonie, 82
most_recent_ats, 135
mutual exclusion problem, 30

n (Anzahl der Applikationsprozesse), 113
Nachbedingung, 27, 28
Nachrichtenlaufzeiten, 41
Nachrichtenredundanz, 108
negotiably(φ) (Definition), 129
Neiger, G., 3, 5
network time protocol, 28
Netzwerkprotokolle, 28
Neumann, P.G., 16
nicht-erreichbare Transition (Definition), 87
nicht-erreichbaren Zustand (Definition), 90
nicht-maskierende Fehlertoleranz, 79
non-masking fault tolerance, 79
Nordahl, J., 79
norm function, 30
Notwendigkeit von Speicherredundanz (Theorem), 90
Notwendigkeit von Zeitredundanz (Theorem), 98
NTP, 28
Null-Nachrichten, 137

observation, 116
observation modality, 117
observation process, 113
observer, 113
offline predicate detection, 114
online predicate detection, 114
optimistische Netzwerkprotokolle, 126
optimistischer Fehlerdetektor (Definition), 126
optimistischer Zustandsverband einer Berechnung (Definition), 129

parity bit, 106
partial-order semantics, 159
partially synchronous communication, 42
partially synchronous processes, 42
partially synchronous systems, 42
partiell synchone Kommunikation, 42
partiell synchrone Prozesse, 42
partiell synchrone Systeme, 42, 53
partielle Korrektheit, 28, 29
Partitionierungen, 53
Paterson, M., 39
Peled, D., 82

perfect failure detector, 49
perfect timing failure detector, 57
perfekter Fehlerdetektor, 49, 121
perfekter Zeitfehlerdetektor, 57
pessimistischer Fehlerdetektor (Definition), 126
pessimistischer Zustandsverband einer Berechnung (Definition), 129
possibly(φ), 117
postcondition, 28
Prädikat, 113
Prädikatstransformator, 117
Präfix eines Ablaufs, 67
precondition, 28
predicate detection problem, 48, 114
predicate transformer, 117
Programm (Definition), 69
Programmverifikation, 27
progress assumptions, 55
$proj_i(ac)$, 124
$proj_i(v)$, 133
$prop(\Sigma)$, 70
propose, 38
Prozeß, 25

qualitative Eigenschaften, 27
quantitative Eigenschaften, 27
quasi synchronous system model, 56
quasi-synchrones Systemmodell, 56
Quicksort, 106

Rabin, M.O., 107
Raum/Zeit-Diagramm, 115
reaktive Systeme, 28
reducibility, 52
redundante Sensoren, 36
Redundanz, 3, 61, 158
Reduzierbarkeit, 52
refinement, 106
Rehabilitierung eines Prozesses nach einem Ereignis durch einen Beobachter, 123
Rehabilitierung eines Prozesses nach einem Ereignis, 126
reliability, 17
reliable broadcast, 39, 40, 51, 52
replizierte Datenbanken, 36
Rushby, J., 80

safety, 17
safety property, 29
Schepers, H., 79
schlechte Transition, 87
schlechte Zustände, 85
Schlichting, R., 34
Schließen mit Zusicherungen, 29
Schneider, F.B., 30, 34, 62, 68, 72, 105
Schnitt, 115
Schwache Fairneß, 71
schwache Genauigkeit, 50
schwache Vollständigkeit, 51
secret sharing, 107
security, 108
Selbststabilisierung, 158
Semantik eines Programms, 70
set decreasing predicates, 118
settled region, 135
Sicherheit, 17
Sicherheitseigenschaft, 29
Sicherheitseigenschaft (Definition), 68
sicherheitskonservatives Fehlermodell (Definition), 83
Sicherheitsspezifikation, 73
Software-Fehlertoleranz, 159
space/time diagram, 115
specification, 26
Speicherredundanz, 63
Speicherredundanz (Definition), 90
Spezifikation, 26
Spezifikation (Definition), 72
Spurengleichheit, 81, 159
stabiler Bereich, 135
starke Fairneß, 71
starke Genauigkeit, 49

starke Vollständigkeit, 49
state machine approach, 62
Stockmeyer, L., 3, 5, 42
strong accuracy, 49
strong completeness, 49
strukturelle Redundanz, 159
sup, 116

temporale Logik, 28
Temporalformeln und deren Semantik (Definition), 71
termination, 38
Terminierung, 28, 29, 38
Terminierungserkennung, 48
Terminierungsfunktion, 30
time-free, 32
time/value, 158
timed asynchronous system model, 54
timing failure, 57
TMR, 67
tolerance specification, 78
top-Element, 116
Toueg, S., 3, 5, 35, 48, 51, 52, 60, 122, 153
trace, 26
Transition, 67
triple modular redundancy, 67, 99

Übereinstimmung, 38
Übereinstimmungsprobleme, 36
unendlich häufig genau, 122
$up[1..n]$, 119
up^k, 119

Validierung, 26
validity, 38
vector clock, 116
Vektoruhr, 116
Venkatesan, S., 118
Verband, 116
Verband über dem erweiterten Zustandsraum, 127
Verdächtigung eines Prozesses nach einem Ereignis, 126
Verdächtigung eines Prozesses nach einem Ereignis durch einen Beobachter, 123
Verfügbarkeit, 17, 36
Verfeinerungsrelation, 106
Verhaltenserweiterung, 75
Verifikation, 26
Verı́ssimo, P., 3, 5, 57
Verklemmungsfreiheit, 29
Verläßlichkeit, 17
verletzbare Spezifikation (Definition), 76
Verletzung einer Spezifikation durch ein Programm, 73
Verletzung einer Eigenschaft durch einen Ablauf, 67
verlustbehaftete Kanäle, 53
Versteigerungen im Internet, 37
verteilte Systeme, 157
verteilter Algorithmus, 26
Völzer, H., 158
Vollständigkeit der Fehlermodellierung (Theorem), 76
Vorbedingung, 27, 28
Vorgänger, 116
voter, 108

Wache, 64
weak accuracy, 50
weak completeness, 51
Weber, D.G., 79
wechselseitiger Ausschlus, 30
witness predicate, 64
Wohlfundiertheitsargument, 30

X-Detektor mittels Z, 64
X-Korrektor mittels Z, 65

zeitüberwachtes Warten, 45
zeitbeschranktes asynchrones Systemmodell, 54
Zeitfehler, 57
zeitfrei, 32
Zeitredundanz, 63

Zeitredundanz (Definition), 98
zusatzliche Fehlerdetektoreigenschaften (Definition), 127
Zusicherung, 27
Zustand, 67
Zustandsübergang über C, 67
Zustandsprädikat über C, 67
Zuverlässigkeit, 17

Übersicht über Lebens- und Bildungsgang

Persönliche Daten

Felix Christoph Gärtner
geboren am 23. Januar 1970 in Korbach

Schulbildung

1974–76	Besuch der *Gladstone Park Infant School, London N.W.2.*
1976–80	Besuch der Grundschule Wittelsberg.
1980–89	Besuch des humanistischen Gymnasium Philippinum, Marburg/Lahn.
Mai 1989	Abitur.
1989–90	Zivildienst im Altenheim St. Elisabeth in Wetter/Hessen.

Hochschulausbildung

Okt. 1990–Feb. 1998	Informatik-Studium an der Technischen Universität Darmstadt mit Nebenfach Germanistik.
Sep. 1993–Aug. 1994	Einjähriger ERSAMUS-Auslandsaufenthalt am Trinity College, Dublin.
Feb. 1998	Abschluß des Studiums an der Technischen Universität Darmstadt mit dem Titel Diplom-Informatiker (Dipl.-Inform.) mit Auszeichnung.
März 1998–Feb. 2001	Promotionsstipendium der Deutschen Forschungsgemeinschaft im Rahmen des Graduiertenkollegs "Intelligente Systeme für die Informations- und Automatisierungstechnik" an der Technischen Universität Darmstadt.