Fehlermodellierung ohne explizite Fehlerzustände

Felix Gärtner, Fachbereich Informatik, Technische Universität Darmstadt

Kurzfassung

Fehler werden häufig durch “virtuelle” Programmänderungen modelliert, zum Beispiel durch Hinzufügung zusätzlicher Variablen und Programmaktionen. Zwar läßt sich auf diese Art praktisch jede denkbare Fehlerannahme formalisieren, jedoch ist gerade die Einführung von expliziten Fehlerzuständen aus theoretischer Sicht her unbefriedigend. Vereinfacht gesagt bedeuten zusätzliche Zustände einen zusätzlichen Abstraktionsschritt, den man bei der theoretischen Untersuchung eines Systems immer dann beachten muß, wenn man fehlerbehaftetes Verhalten mit fehlerfreiem Verhalten in Beziehung setzt. Wir stellen ein Verfahren vor, welches es erlaubt, praktisch jede Fehlerannahme *ohne* zusätzliche Systemzustände zu formalisieren. Das Verfahren basiert auf einer klaren Trennung von Sicherheits- und Lebendigkeitseigenschaften und erlaubt, einfacher über fehlertolerante Systeme zu schließen als dies mit bisherigen Methoden möglich war.

1 Einführung

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 beinhaltet Ereignisse, die bei der Ausführung von programmierten, d.h. “erwarteten” Anweisungen entstehen. Die zweite Klasse besteht aus “unerwarteten” Fehlereignissen. Man kann also Fehlverhalten auch in Form eines Programmes ausdrücken [6, 8]. Beispielsweise ist es relativ einfach, die Fehler in einem 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.[1]

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 der Praxis 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* [11, 17].

Man unterscheidet gewöhnlich die *lokale* und 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* [11, 9]. In der Literatur existiert eine Vielzahl von unterschiedlichen Fehlerannahmen. Im Bereich der fehlertoleranten verteilten Algorithmen sind zwei Fehlerannahmen besonders populär:

1. *Fail-stop* [24]: Die einzigen Fehler, die auftreten können, sind sogenannte *Anhalteausfälle*, d.h. einzelne Rechnerknoten können spontan aufhören, ihr lokales Programm abzuarbeiten. Der Einfachkeit halber nimmt man an, daß diese Rechner auf ewig abgestürzt bleiben. Diese Fehlerannahme approximiert das aus der Realität bekannte “Hängen” eines Rechners, etwa wegen eines dauerhaften Hardwaredefekts. Zusätzlich zu dieser lokalen Fehlerannahme gibt die globale Fehlerannahme eine obere Schranke für die Anzahl der so in Mitleidenschaft gezogenen Rechnerknoten.

2. *Arbitrary transient fault* [7]: Hier geht man da-

[1] Nach Rushby [23] entspricht dies einer *berechnenden* Herangehensweise an das Problem (*calculational*). Die Alternative sind *spezifikationsbasierte* (*specificational*) Ansätze, die Fehlverhalten auf einer semantischen Ebene, d.h. mittels abgeschwächter Komponentenspezifikation modellieren [16, 10]. Wir beschränken uns in diesem Beitrag auf berechnende Methoden.

von aus, daß einzelne oder alle Variablen eines Programms spontan einen neuen Wert annehmen können (lokale Fehlerannahme). Diese Fehlerannahme modelliert die aus der Realität bekannten Auswirkungen von kosmischer Strahlung auf den Inhalt von Speicherchips. Natürlich muß man sinnvollerweise annehmen, daß derartige Fehler nur endlich oft oder in bestimmten Mindestabständen auftreten (globale Fehlerannahme).

Es gibt verschiedene Formalismen, die bisher versuchten, diese (und andere) Fehlerannahmen zu präzisieren [5, 4, 25, 13]. Sie alle basieren auf zwei Prinzipien: (1) Einführung zusätzlicher Programmzustände (z.B. durch zusätzliche Variablen) und (2) Einführung zusätzlicher Zustandsübergänge (z.B. in Form von zusätzlichen Programmaktionen). Zum Beispiel wird *Fail-stop* gewöhnlich modelliert, indem ein hypothetischer "Absturzzustand" eingeführt wird, in den das System unter bestimmten Bedingungen (durch Einführung von neuen Transitionen) spontan wechseln kann.[2] Es ist leicht einzusehen, daß diese Art der Fehlermodellierung *vollständig* ist, d.h. man kann jede Fehlerannahme, die man genau genug beschreiben kann, mittels dieser zwei Prinzipien modellieren.

Die Einführung zusätzlicher Zustände ist aus theoretischer Sicht nicht immer zu begrüßen. Oft möchte man nämlich Aussagen darüber machen, wie sich das Verhalten eines Systems Σ unter Fehlereinflüssen ändert. Genauer gesagt möchte man das Verhalten von Σ, d.h. dem originalen, fehlerfreien System, in Beziehung setzen zum Verhalten von $F(\Sigma)$, d.h. dem System, welches der Fehlerannahme F ausgesetzt ist. Wenn F jedoch zusätzliche Zustände in Σ einführt, ist das Verhalten beider Systeme jedoch nicht einfach vergleichbar, da Σ und $F(\Sigma)$ unterschiedliche Zustandsräume besitzen. Man benötigt eine Projektionsfunktion ϕ, die Zustände von $F(\Sigma)$ abbildet auf entsprechende Zustände von Σ. Wenn man Verhalten als eine Folge von Zuständen definiert, führt aber die Einführung einer solchen Funktion zu neuen Problemen: Die Projektion $\phi(\sigma)$ einer Zustandsfolge σ kann aufgrund von "verdeckten" Fehlerzuständen ein sogenanntes *Stottern* (*stuttering*) enthalten, d.h. eine Folge von gleichen Systemzuständen von Σ, die im originalen System unter Umständen nicht erlaubt waren. Alle diese Probleme sind wohlbekannt aus der Theorie der *Verfeinerung* (*refinement*) [1], einer formalen Entwurfs- und Implementierungstechnik, die auf schrittweisem Konkretisieren einer abstrakten Spezifikation beruht.

Wenn man fehlertolerante Systeme theoretisch untersucht, sind zusätzliche Systemzustände demnach nicht sonderlich wünschenswert. Aus anderen Bereichen der theoretischen Informatik wissen wir jedoch, daß man das Verhalten von Systemen in bestimmten Fällen auch beinflussen kann, ohne zusätzliche Zustände einzuführen. Beispielsweise kann man in der Theorie nebenläufiger Systeme die Terminierung eines Algorithmus erst dann beweisen, wenn man annimmt, daß die nebenläufigen Aktivitätsträger *fair* behandelt werden, d.h. in einer gerechten Reihenfolge die Schritte ihrer Programme abarbeiten. Formal gesagt, kann man oft ohne zusätzliche Zustände auskommen, wenn man *Lebendigkeitseigenschaften* mittels *Lebendigkeitsannahmen* beeinflussen kann. Lebendigkeitseigenschaften fordern, daß bestimmte Dinge nach endlicher Zeit oder unendlich oft stattfinden. Beispiele sind die Terminierung oder die Aushungerungsfreiheit eines Algorithmus. Auf der anderen Seite beschreiben Lebendigkeitsannahmen die grundsätzlichen Systemannahmen über die Aktivität eines Algorithmus. Beispiele sind starke und schwache Fairness [12] oder Maximalität (d.h. wann immer noch ein Zustandsübergang möglich ist, wird nach endlicher Zeit irgendein solcher Übergang ausgeführt).

Dieser Beitrag untersucht, inwiefern sich diese Ergebnisse auch für die Vereinfachung der Fehlermodellierung nutzbar machen lassen. Die Frage lautet: Kann man Fehlerannahmen vollständig modellieren, ohne zusätzliche Zustände einführen zu müssen? Unter bestimmten Voraussetzungen, die wir im folgenden kurz umreißen, kann man diese Frage bejahen.

Die erste Voraussetzung besagt, daß wir nur *zeitfreies* Fehlverhalten betrachten. Zeitfrei bedeutet, daß es unabhängig von Echtzeitbedingungen (beispielsweise mittels temporaler Logik) beschrieben werden kann. Wenn man der Argumentation von Abadi und Lamport [2] folgt, ist dies aber keine theoretische Einschränkung, da man durch explizite Modellierung einer Uhr im System auch Echtzeitverhalten spezifizieren kann.

Zweitens schränken wir unsere Betrachtung ausschließlich auf *lokale* Fehlerannahmen ein, also auf das Fehlverhalten individueller Prozesse. Globale Fehlerannahmen (wie zum Beispiel: "maximal k Prozesse stürzen ab") beschränken die Entfaltungsmöglichkeit lokaler Fehlerannahmen in Abhängigkeit von einem unter Umständen nicht beobachtbaren Systemzustand (beispielsweise dem Absturz von Prozessen). Im allgemeinen müssen globale Fehlerannahmen darum zusätzliche Variablen (und damit Zustände) einführen.

Mit der eben gemachten Beobachtung hängt auch die dritte Einschränkung zusammen: Fehlverhalten muß *gedächtnislos* sein, d.h. in einem gegebenen Systemzustand tritt ein Fehler unabhängig davon auf, ob vorher bereits Fehler stattgefunden haben oder nicht. Ein Beispiel für "gedächtnisbehaftetes" Fehlverhalten ist, wenn in einer Programmschleife ein transienter Fehler höchstens ein Mal auftreten darf. Gedächtnislos ist das Fehlverhalten, wenn der Fehler in *jedem* Schleifendurchlauf auftreten kann oder nicht. Wie die Umsetzung von globalen Fehlerannahmen ist gedächtnisbehaftetes Fehlverhalten im allgemeinen also nicht ohne zusätzlicher Variablen zu erreichen.

Wir stellen in den Kapiteln 2 und 3 einen einfachen und

[2] Natürlich kann man einen Absturz auch durch eine Endlosschleife modellieren, also ohne einen zusätzlichen Zustand. In der Regel wird dies jedoch als ungenügend empfunden.

doch allgemeinen Formalismus vor, der Spezifikationen, Programme und Fehlerannahmen ohne zusätzliche Fehlerzustände präzisiert. Anhand von Beispielen und mittels eines einfachen Vollständigkeitsresultates zeigen wir dann, daß man jede Fehlerannahme, die zur Verletzung einer gegebenen Spezifikation führen könnte, mittels der vorgestellten Definitionen formalisieren kann. Die Grundlage des Formalismus liegt in einer Trennung des Verhaltens in einen *Sicherheitsanteil* (*safety*) und einen *Lebendigkeitsanteil* (*liveness*) [18]. Eine Verletzung der *safety* ist (bis auf triviale Fälle) unter den gegebenen Voraussetzungen immer durch zusätzliche Transitionen erreichbar. Auf der anderen Seite ist die Verletzung der *liveness* durch eine Abschwächung der Lebendigkeitsannahme möglich. Aus der vollständigen Zerlegbarkeit jeder Spezifikation in Sicherheits- und Lebendigkeitsanteil [3] folgt die Vollständigkeit unserer Methode.
Aus unserer Erfahrung bedeutet die Modellierung von Fehlern ohne zusätzliche Zustände eine deutliche Vereinfachung bei der theoretischen Analyse fehlertoleranter verteilter Algorithmen. Beispielsweise resultieren grundlegende Resultate über die Art und Rolle von Redundanz in fehlertoleranten Systemen [14, 15] aus der vereinfachenden Betrachtungsweise des hier vorgestellten Formalismus. Wir vermuten den Grund hierfür darin, daß mit unserer Methodik deutlicher als zuvor die unterschiedlichen und theoretisch fundamentalen Aspekte von *Sicherheit* und *Lebendigkeit* separiert werden können. Da die genannten Resultate zur Redundanz bereits dokumentiert sind [14] wird in einem abschließenden Kapitel 4 auf die Bezüge dieser Arbeit zu anderen grundlegenden Aussagen zur Fehlermodellierung eingegangen.

2 Programme, Spezifikationen und Korrektheit

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 (endliche oder 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.

Definition 1 (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:*

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

Definition 2 (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 1 charakterisiert also eine Menge von "unerwünschten" endlichen Abläufen. Im Gegensatz dazu besagt Definition 2, daß Lebendigkeitseigenschaften nur im Unendlichen verletzt werden können. Eine Lebendigkeitseigenschaft schließt also eine Menge von Ablaufsuffixen aus. Alpern und Schneider [3] 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 1** ein Programm mit vier Zuständen a, b, c und d. Startzustände sind durch einen Stern gekennzeichnet und Zustandsübergänge durch einen Pfeil.

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* [26] bezeichnen.

Definition 3 (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.*

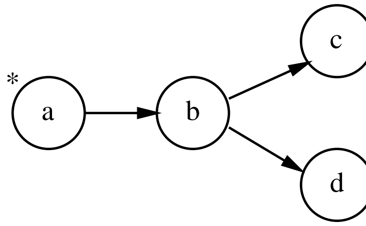


Abbildung 1: Beispiel für ein einfaches Programm.

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 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 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 in c oder in d enden. Die folgenden Abläufe sind Beispiele für Elemente von A:

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

Definition 4 (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.*

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

Fairneß- und Lebendigkeitsannahmen. Viele der in der Literatur bekannten *Fairneßannahmen* sind Lebendigkeitsannahmen. Fairneßannahmen werden in der Theorie nebenläufiger Systeme benötigt, um die *faire Konfliktauflösung* wiederkehrend konkurrierender Transitionen zu regeln. Beispielweise zeigt **Abbildung 2** ein Programm, bei der Zustand 2 wiederholt eingenommen werden kann. In diesem Zustand hat das System außerdem die Wahl zwischen drei verschiedenen "konkurrierenden" Transitionen. Damit das System letztendlich auch einmal Zustand 3 erreicht, reicht Maximalität als Lebendigkeitsannahme nicht aus, denn Maximalität erlaubt beispielsweise, daß das System unendlich zwischen den Zuständen 1 und 2 hin- und herspringt.
Es gibt in der Literatur zwei prominente Arten von Fairneß [12], die wir in unser Systemmodell übertragen:

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

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

Jeder stark faire Ablauf ist auch schwach fair. Schwache Fairneß sondert Abläufe aus, in denen eine Transition nicht ausgeführt wird, obwohl sie unendlich oft *hintereinander* möglich war. In unserem Beispiel ist der Ablauf $x = 1, 2, 2, 2, \ldots$ nicht schwach fair. Bei starker Fairneß muß die Transition nicht unendlich lange hintereinander möglich sein, sondern es reicht aus, wenn sie unendlich oft ausführbar ist (sie kann auch zwischendurch einmal nicht möglich sein). In unserem Beispiel ist der Ablauf $x = 1, 2, 1, 2, 1, 2, \ldots$ nicht stark fair. Gleiches gilt für den Ablauf $x = 1, 2, 2, 1, 2, 2, 1, \ldots$, denn hier wird die Transition $(2, 3)$ unfair behandelt. Nur terminierende Abläufe erfüllen hier die starke Fairneß. Starke und schwache Fairneß sind Lebendigkeitsannahmen im Sinne von Definition 3.
Eine Lebendigkeitsannahme ist umso stärker, je mehr "unfaire" Abläufe sie ausschließt. Starke Fairneß ist demnach stärker als schwache Fairneß, und schwache Fairneß ist stärker als Maximalität. Die schwächste Lebendigkeitsannahme ϵ sondert keinerlei Abläufe aus.

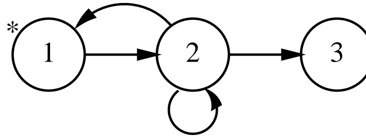


Abbildung 2: Ein einfaches Programm.

Transitionsgebundene Lebendigkeitsannahmen. Bisher waren in einem gegebenen System Lebendigkeitsannahmen global, d.h. sie galten für alle Transition in gleicher Weise. Man kann aber Lebendigkeitsannahmen auch auf einzelne Transitionen beziehen. Als Beispiel betrachten wir wieder Abbildung 2 und nehmen an, daß alle Transitionen außer der Transition (2,3) stark fair behandelt werden. Für die Transition (2,3) gelte Maximalität, d.h. wenn sich das System in Zustand 2 befindet, muß irgendeine Transition genommen werden (nicht notwendigerweise die Transition von 2 nach 3). In diesem Fall ist genauso wie bei "globaler" starker Fairneß der Ablauf $1 \cdot 2 \cdot 2 \cdot 2 \cdots$ nicht erlaubt. Erlaubt ist hingegen nun der Ablauf $1 \cdot 2 \cdot 2 \cdot 1 \cdot 2 \cdot 2 \cdot 1 \cdots$, der vorher als nicht stark fair ausgesondert war. Mit der Abschwächung der Lebendigkeit ist das Erreichen von Zustand 3 nun nicht mehr garantiert.
Da man Lebendigkeitsannahmen schwer visualisieren kann, werden wir in den Abbildungen von Programmen die zum Programm gehörige Lebendigkeitsannahme (also z.B. Maximalität) als Text entweder global oder an den Transitionen annotieren.

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 [4] beschränken wir uns auf sogenannte *fusions-abgeschlossene Eigenschaften* (*fusion closed properties*). Fusionsabgeschlossenheit ist die Formalisierung der eingangs erwähnten Gedächtnislosigkeit des Fehlverhaltens.

Definition 5 (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 5 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**). Ein Beispiel für eine Eigenschaft, die nicht fusions-abgeschlossen ist, lautet: "Zustand x tritt nur auf falls vorher irgendwann Zustand y aufgetreten ist."

Definition 6 (Spezifikation) *Eine* Spezifikation über C *ist eine fusions-abgeschlossene Eigenschaft SPEC über C.*

In Anlehnung an Alpern und Schneider [3] 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*.
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 Fehlermodellierung ohne explizite Fehlerzustände

Wir greifen die eingangs besprochene Idee auf und modellieren Fehler als unerwünschtes, zusätzliches Programmverhalten.

Definition 7 (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. $I' \supseteq I$
3. $T' \supseteq T$
4. $A' \supseteq A$

Gilt $F(\Sigma) = \Sigma'$ wird Σ' die *fehlerbehaftete Version* von Σ genannt. Manchmal bezeichnen wir mit $F(I)$, $F(T)$ und $F(A)$ die Mengen I', T' und A' der fehlerbehafteten Version.
In unserer Definition fügen Fehlermodelle keine neuen Zustände hinzu. Fehler können aber dazu führen, daß das System in einem "unerlaubten" Zustand beginnt, d.h. einem Zustand, der vorher kein Startzustand war (Bedingung 2). Dies modelliert Fehler, die vor dem Einschalten des Systems auftreten. Außerdem können Fehler 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.

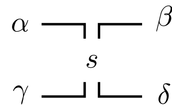


Abbildung 3: Schaubild zur Fusions-Abgeschlossenheit.

Abbildung 4 zeigt ein einfaches Programm und dessen fehlerbehaftete Version. Angenommen die Spezifikation *SPEC* für Σ besteht aus zwei Teilen, einem Sicherheitsanteil *SSPEC* $\equiv$"Zustand c wird ausschließlich aus Zustand b erreicht", und einem Lebendigkeitsanteil *LSPEC* $\equiv$"Nach endlicher Zeit wird c erreicht". Das Fehlermodell hat zwei zusätzliche Zustandsübergänge eingefügt, die als gestrichelte Pfeile dargestellt sind, und hat die Lebendigkeitsannahme global auf Maximalität abgeschwächt. Offensichtlich wird durch die neue Transition (a, c) *SSPEC* verletzt. Durch die Abschwächung der Lebendigkeitsannahme wird zudem erreicht, daß der Ablauf $a \cdot b \cdot b \cdots$ nicht mehr unfair ist, so daß auch *LSPEC* verletzt wird.

Die Fehlermodellierung durch Definition 7 ist in einem gewissen Sinne vollständig: Man kann jede Spezifikation durch ein Fehlverhalten verletzen, das durch eine Funktion F "implementiert" wird. Eine *verletzbare Spezifikation* ist eine Spezifikation *SPEC*, die eine echte Untermenge von C^* ist, also der Menge aller Abläufe, die mit Zuständen aus C gebildet werden können. Natürlich läßt sich immer ein triviales "schlimmstes" Fehlermodell angeben, welches durch die Wahl von $F(I) = C$, $F(T) = C \times C$ und $F(A) = \epsilon$ entsteht. In dem Theorem geht es uns jedoch um den Nachweis der Existenz eines nichttrivialen Fehlermodells.

Theorem 1 (Vollständigkeit der Fehlermodellierung) *Sei SPEC eine verletzbare Spezifikation und Σ ein Programm, welches SPEC erfüllt. Dann existiert ein nichttriviales F, so daß $F(\Sigma)$ die Spezifikation SPEC verletzt.*

Die Darstellung des Beweises erfolgt in einer strukturierten Notation ähnlich der von interaktiven Theorembeweisern. Diese Form wurde von Lamport [19] 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 (Teil-)Beweises jeweils eine Beweisskizze gegeben.

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: Wir konstruieren schrittweise ein Fehlermodell F, mit dem man einen Ablauf nachspielen kann, der nicht in *SPEC* liegt. Verletzungen der Sicherheit kann man dabei durch die Hinzunahme von Zustandsübergängen erreichen, während Verletzungen der Lebendigkeit durch eine Kombination von zusätzlichen Zustandsübergängen und zusätzlichen "unfairen" Abläufen erreicht werden können.

$\langle 1 \rangle 1.$ Es existiert ein Ablauf $\sigma \in C^*$, der nicht in *SPEC* liegt.

BEWEIS: Existenz folgt aus Voraussetzung, daß *SPEC* eine verletzbare Spezifikation ist (eine *echte* Teilmenge von C^*). □

$\langle 1 \rangle 2.$ Bezeichne *SSPEC* den Sicherheits- und *LSPEC* die Lebendigkeitsanteil von *SPEC*. Dann gilt $\sigma \notin$ *SSPEC* oder $\sigma \notin$ *LSPEC*.

BEWEIS: Folgt aus Schritt $\langle 1 \rangle 1$ und dem Theorem von Alpern und Schneider [3]. □

$\langle 1 \rangle 3.$ ANNAHME: $\sigma \notin$ *SSPEC*
ZEIGE: Q.E.D.

BEWEISIDEE: Hier genügt es, entweder einen "schlechten" Anfangszustand in I oder eine bestimmte "schlechte" Transition in T aufzunehmen. Im letzteren Fall wird die Existenz einer solchen Transition durch die Fusionsabgeschlossenheit von *SSPEC* garantiert. Für eingefügte Transition muß Maximalität als Lebendigkeitsannahme angenommen werden.

$\langle 2 \rangle 1.$ ANNAHME: σ hat einen Anfangszustand s, der nicht in I enthalten ist.
ZEIGE: Q.E.D

BEWEIS: Definiere $F(I) = I \cup \{s\}$. □

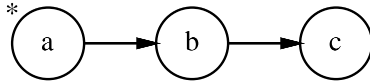
$\langle 2 \rangle 2.$ ANNAHME: Der Startzustand von σ ist in I enthalten.
ZEIGE: Q.E.D.

$\langle 3 \rangle 1.$ Es existiert eine Transition $t \in T$, für die gilt, daß alle Abläufe, in denen t vorkommt, *SSPEC* verletzen.

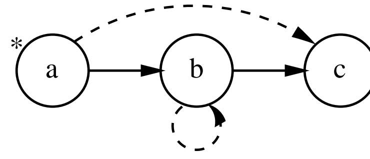
$\langle 4 \rangle 1.$ σ kann geschrieben werden als $\alpha \cdot d \cdot b \cdot \beta$ so daß $\alpha \cdot d$ zu einem Ablauf in *SSPEC* fortgesetzt werden kann, nicht aber $\alpha \cdot d \cdot b$.

BEWEIS: Die Existenz von d folgt aus der Fallannahme, daß der Startzustand von σ ein erlaubter Anfangszustand in I ist. Der Rest folgt aus der Tatsache, daß *SSPEC* eine Sicherheitseigenschaft. □

$\langle 4 \rangle 2.$ Betrachte einen anderen Ablauf ρ von Σ, in



Σ (A = schwache Fairness)



$F(\Sigma)$ (A = Maximalität)

Abbildung 4: Beispiel für ein Programm und dessen fehlerbehaftete Version. Die Lebendigkeitsannahme von $F(\Sigma)$ wurde abgeschwächt auf Maximalität.

dem die Transition (d, b) auftritt, also $\rho = \hat{\alpha} \cdot d \cdot b\hat{\beta}$. Dann gilt $\rho \notin$ *SSPEC*.

$\langle 5 \rangle 1$. ANNAHME: $\rho \in$ *SSPEC*
ZEIGE: falsch

$\langle 6 \rangle 1$. Alle Präfixe von ρ lassen sich zu Abläufen in *SSPEC* fortsetzen.
BEWEIS: Folgt aus der Annahme von Schritt $\langle 5 \rangle 1$ und der Voraussetzung, daß *SSPEC* eine Sicherheitseigenschaft ist. □

$\langle 6 \rangle 2$. $\hat{\alpha} \cdot d \cdot b$ läßt sich zu einem Ablauf in *SSPEC* fortsetzen.
BEWEIS: Folgt aus Schritt $\langle 6 \rangle 1$ und der Tatsache, daß $\hat{\alpha} \cdot d \cdot b$ ein Präfix von ρ ist. □

$\langle 6 \rangle 3$. $\exists \hat{\delta} : \hat{\alpha} \cdot d \cdot b \cdot \hat{\delta} \in$ *SSPEC*.
BEWEIS: Folgt aus Schritt $\langle 6 \rangle 2$. □

$\langle 6 \rangle 4$. $\exists \delta : \alpha \cdot d \cdot \delta \in$ *SSPEC*.
BEWEIS: Folgt aus Schritt $\langle 4 \rangle 1$ ($\alpha \cdot d$ kann zu einem Ablauf in *SSPEC* fortgesetzt werden). □

$\langle 6 \rangle 5$. $\alpha \cdot d \cdot b \cdot \hat{\delta} \in$ *SSPEC*.
BEWEIS: Folgt aus Schritten $\langle 6 \rangle 3$ und $\langle 6 \rangle 4$ und der Fusionsabgeschlossenheit von *SSPEC*. □

$\langle 6 \rangle 6$. $\alpha \cdot d \cdot b$ kann zu einem Ablauf in *SSPEC* fortgesetzt werden.
BEWEIS: Folgt aus Schritt $\langle 6 \rangle 5$. □

$\langle 6 \rangle 7$. Q.E.D.
BEWEIS: Schritt $\langle 6 \rangle 6$ ergibt einen Widerspruch zu Schritt $\langle 4 \rangle 1$ (daß nämlich $\alpha \cdot d \cdot b$ *nicht* zu einem Ablauf in *SSPEC* fortgesetzt werden kann). □

$\langle 5 \rangle 2$. Q.E.D.
BEWEIS: Folgt indirekt aus Schritt $\langle 5 \rangle 1$. □

$\langle 4 \rangle 3$. Q.E.D.
BEWEIS: Nehme die Transition $t = (d, b)$ aus Schritt $\langle 4 \rangle 1$. Schritt $\langle 4 \rangle 2$ zeigt, daß jede Ablauf, der t enthält, die Spezifikation verletzt. □

$\langle 3 \rangle 2$. Q.E.D.
BEWEIS: Definiere $F(T) = T \cup \{t\}$. □

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

$\langle 1 \rangle 4$. ANNAHME: $\sigma \notin$ *LSPEC*
ZEIGE: Q.E.D.

BEWEISIDEE: Hier reicht es aus, den unerlaubten Suffix von σ nachzuspielen. Dies kann durch die Abschwächung der Lebendigkeitsannahme und ggf. durch das Hinzufügen neuer Transitionen geschehen.

$\langle 2 \rangle 1$. Es existiert ein nichtleerer Suffix β von σ, mit dem alle endlichen Abläufe von Σ nicht in *LSPEC* liegen.
BEWEIS: Folgt aus der Fallannahme ($\sigma \notin$ *LSPEC*) und der Tatsache, daß *LSPEC* eine Lebendigkeitseigenschaft ist. □

$\langle 2 \rangle 2$. ANNAHME: Alle Transitionen aus β sind in T enthalten.
ZEIGE: Q.E.D.
BEWEIS: Definiere $F(A)$ als die stärkste Lebendigkeitsannahme, die $A \cup \{\sigma\}$ enthält. □

$\langle 2 \rangle 3$. ANNAHME: Es gibt Transitionen aus β, die nicht in T enthalten sind.
ZEIGE: Q.E.D.
BEWEIS: Füge in $F(T)$ alle solche Transitionen hinzu und definiere $F(A)$ als die stärkste Lebendigkeitsannahme, die $A \cup \{\sigma\}$ enthält. □

$\langle 2 \rangle 4$. Q.E.D.
BEWEIS: Schritte $\langle 2 \rangle 2$ und $\langle 2 \rangle 3$ decken alle Fälle ab. □

$\langle 1 \rangle 5$. Q.E.D.
BEWEIS: Schritte $\langle 1 \rangle 3$ und $\langle 1 \rangle 4$ decken laut Schritt $\langle 1 \rangle 2$ 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 1 ist konstruktiv in dem Sinne, daß man zu einem Programm und einer verletzbaren Spezifikation quasi ein "minimales" Fehlermodell angeben kann, welches die Spezifikation verletzt. Als Beispiel betrachten wir den Automaten in **Abbildung 5** (links), der im fehlerfreien Fall die Spezifikationen *SSPEC* ≡"niemals f" und *LSPEC* ≡"unendlich oft 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 der Transition (c, d) auf Maximalität abzuschwächen. Jetzt ist ein

unendliches Verweilen (*livelock*) in Zustand c möglich, und damit die Verletzung von *LSPEC* gegeben. Alternativ könnte man auch die Lebendigkeitsannahme von (c, d) auf ϵ abschwächen. Dadurch kann ein echtes Stehenbleiben (*crash*) in Zustand c modelliert werden.

4 Abschließende Bemerkungen

In seiner Dissertation gibt Masum [21] einen umfassenden Überblick über bisherige Ansätze zur Fehlermodellierung. Die Mehrheit der besprochenen Arbeiten, zu denen auch die fundamentalen Beiträge von Laprie [20] und Powell [22] gehören, basiert auf der Unterscheidung zweier Fehlerdomänen: Wertefehler (*value failures*) und Zeitfehler (*timing failures*). Der in diesem Beitrag vorgestellte Ansatz spiegelt diese Unterscheidung im Kontext zeitfreier Probleme wieder: Wertefehler entsprechen hierbei einer Verletzung der Sicherheit, während sich Zeitfehler in der Verletzung der Lebendigkeit äußern. Bemerkenswert ist jedoch, daß aus dieser Perspektive nur Endlosschleifen und *fail-stop*-Fehler als *timing failures* betrachtet werden können. Zeitfehler machen also auch im Kontext zeitfreier Probleme Sinn.

Danksagungen

Der Autor bedankt sich bei Hagen Völzer für klärende Diskussionen sowie die vielfältigen und hilfreichen Verbesserungsvorschläge zu diesem Beitrag.

Literatur

[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] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[4] Anish Arora and Sandeep S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

[5] Max Breitling. Modeling faults of distributed, reactive systems. 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 58–69, Pune, India, September 2000. Springer-Verlag.

[6] Flaviu Cristian. A rigorous approach to fault-tolerant programming. *IEEE Transactions on Software Engineering*, 11(1):23–31, January 1985.

[7] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[8] 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.

[9] Klaus Echtle. *Fehlertoleranzverfahren*. Springer-Verlag, 1990.

[10] 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.

[11] 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.

[12] Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[13] 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.

[14] Felix C. Gärtner. *Formale Grundlagen der Fehlertoleranz in verteilten Systemen*. PhD thesis, Fachbereich Informatik, TU Darmstadt, May 2001.

[15] Felix C. Gärtner and Hagen Völzer. Defining redundancy in fault-tolerant computing. In *Brief Announcement at the 15th International Symposium on DIStributed Computing (DISC 2001)*, Lisbon, Portugal, October 2001.

[16] Maurice P. Herlihy and Jeannette M. Wing. Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93–104, January 1991.

[17] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, April 1997.

[18] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[19] Leslie Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, August/September 1995.

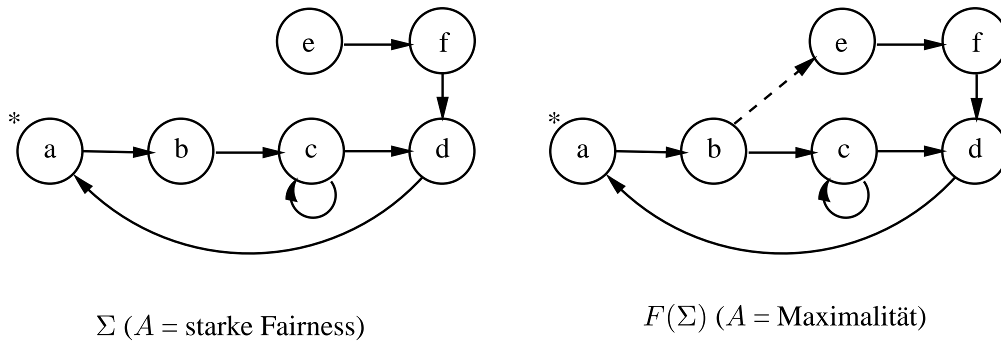


Abbildung 5: Beispiel zur Konstruktion von F. Die Lebendigkeitsannahme von Σ ist starke Fairneß, die von $F(\Sigma)$ ist zu Maximalität abgeschwächt worden.

[20] Jean-Claude Laprie, editor. *Dependability: Basic concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, 1992.

[21] 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.

[22] 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.

[23] John Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and System Safety*, 43(2):189–219, 1994.

[24] 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.

[25] Hagen Völzer. Verifying fault tolerance of distributed algorithms formally: An example. In *Proceedings of the International Conference on Application of Concurrency to System Design (CSD98)*, pages 187–197, Fukushima, Japan, March 1998. IEEE Computer Society Press.

[26] Hagen Völzer. *Fairness, Randomisierung und Konspiration in verteilten Algorithmen*. PhD thesis, Humboldt Universität zu Berlin, Fakultät für Informatik, December 2000.