

Dependable Systems

Ausarbeitung der gesammelten Prüfungsfragen

22. Juni 2014

Inhaltsverzeichnis

2. Kapitel (Basic concepts and terminology)	2
2.1. Was bedeutet Dependability?	2
2.2. Dependability Attribute (Parameter)	3
2.3. Reliability vs Safety	4
2.4. Reliability vs Availability	4
2.5. Wie hoch ist die Availability in einem System, das nicht gewartet wird?	4
2.6. Spezifikation	4
3. Kapitel (Fault-tolerance and Modeling)	6
3.1. Warum werden beim Modellieren von zuverlässigen Systemen gerne konstante Fehlerraten angenommen?	6
3.2. Wahrscheinlichkeitstheoretische Modellierung was gehört dazu?	6
3.3. Simple/ Arbitrary Block Diagram, Markov Model, General Stochastic Petri Net?	7
3.3.1. Simple Block Diagram	7
3.3.2. Arbitrary Block Diagram	8
3.3.3. Markov Modell	8
3.3.4. GSPN General Stochastic Petri Net	10
3.4. Markoveigenschaft	10
3.5. Modellierung für Software	10
3.6. Reliability Growth Model?	11
3.7. Assumption Coverage	11
3.8. Was ist Coverage allgemein und was bedeutet sie beim MarkovGraphen?	12
4. Kapitel (Processes and Certification Standards)	13
4.1. Probability/Severity	13
5. Kapitel (Failure modes and models)	14
5.1. Failure Modes?	14

5.2. Arrhenius equation	15
5.3. Accelerated Stress Testing?	15
5.4. Safety-Analysis	16
6. Kapitel (System aspects of dependable computers)	18
6.1. Maintenance (warum besser als FaultTolerance)?	18
6.2. Systematic VS app-specific fault-tolerance?	19
6.3. Systematic VS app-specific fault-tolerance?	20
6.4. Replica Determinismus?	21
6.5. Synchrone vs Asynchrone Systeme	22
6.6. Was ist konsistenz?	22
6.7. Was ist konsistenz bei Replica?	22
6.8. Consensus Protocols?	22
6.9. Reliable Broadcast	23
6.10. Lässt sich in jedem System Consensus herstellen?	23
6.11. Consistency Algorithm	23
6.12. Common Mode Failure? Diagnose?	24
6.13. Recovery Lines?	24
6.14. Fault Tolerance by Self-Stabilization	24
6.15. Wie viele Replikas um Crash/Performance/Byzantine Failure zu tolerieren (+ warum 3k+1 bei byzantine und nicht 2k+1)	25
6.16. Fail-Silent/Fail Consistent	25
6.17. Single Point of Failure	25
A. Zusatzinfos	26
A.1. Kapitel:	26
A.1.1. Gründe für niedriges Vertrauen	26
A.2. Kapitel:	26
A.2.1. Mögliche Requirements	26
A.2.2. Beeinträchtigung der Zuverlässigkeit(Dependability):	26
A.2.3. Fehlerklassen	27
A.2.4. Fehlerzustand -> Ausfall	27
A.2.5. Mittel der Systemverlässlichkeit	27
A.2.6. Fehlertoleranz und redundanz	28

2. Kapitel (Basic concepts and terminology)

2.1. Was bedeutet Dependability?

7.2013, Ausarb. 2009

Dependability¹ ist die Verlässlichkeit eines Computersystems, so dass es vertretbar ist sich auf die zur Verfügung gestellten Services zu verlassen.

Reliance²: Man kann sich darauf verlassen, dass

¹Dependability: Systemstabilität, Zuverlässigkeit

²Reliance: Verlass, Vertrauen

- sich das System gemäß den Spezifikationen verhält
- das System Hazards³ (=Verhalten, das zu unerwünschten Konsequenzen führt) verhindert

Je näher die Servicespezifikation und das Risiko(Hazard) sind, desto höher ist der Gefährlichkeitsgrad des Systems. \Rightarrow Anwendungsspezifischer Fehlertoleranzrahmen.

2.2. Dependability Attribute (Parameter)

7.2013, Ausarb. 2009

Reliability (Funktionsfähigkeit)

Verlässlichkeit bezogen auf Kontinuität des Service. $R(t)$ ist die Wahrscheinlichkeit, dass das System über eine Periode t hinweg ununterbrochen zur Verfügung steht.

$$R(0) = 1, R(\infty) = 0$$

Bei einer konstanten Fehlerrate sieht sie wie folgt aus:

$$R(t) = e^{-\lambda t}$$

Availability (Verfügbarkeit)

Verlässlichkeit bezogen auf die Verfügbarkeit zu einem bestimmten Zeitpunkt: A oder V ist der Prozentsatz der Zeit, zu dem das System gemäß den Spezifikationen funktioniert.

Safety (Sicherheit)

Verlässlichkeit bezogen auf das Vermeiden katastrophaler Auswirkungen: $S(t)$ ist die Wahrscheinlichkeit, dass ein bestimmtes katastrophales Verhalten innerhalb einer Periode t nicht auftritt.

Die nicht Erreichbarkeit von Zuständen, die zu Tod, Verletzung, Schaden, Berufskrankheit, oder zu einem Verlust von Gerät oder Besitz führen. Safety ist ein relativer Begriff.

Security (Vertraulichkeit)

Verlässlichkeit bezogen auf das Verhindern von unerlaubtem Zugriff auf Informationen.

- Secrecy⁴: wer kann Daten lesen
- Integrity⁵: wer kann Daten ändern und wie?
- Availability⁶: für wen ist es möglich, die Verfügbarkeit eines Systems zu verringern

Weitere:

Usability (Verwendbarkeit), Recoverability (Wiederherstellbarkeit), Maintainability (Wartbarkeit), Extendability (Erweiterbarkeit), ...

³Hazard: Gefahr, Riskio

⁴secrecy: Geheimhaltung

⁵integrity: Integrität, Unbescholtenheit

⁶availability: Verfügbarkeit

2.3. Reliability vs Safety

7.2013, Ausarb. 2009

Reliability wird durch die Verlässlichkeit des kontinuierlichen Services und der funktionalen Spezifikation des Services beschrieben, Safety wird durch eine nicht funktionale Spezifikation beschrieben. Sie ist die Verlässlichkeit der Verhinderung von Katastrophalen Konsequenzen. Das bedeutet, die Verhinderung von **Hazards**.

$$\text{Safety} \subseteq \text{Reliability}$$

Die Auswirkungen einer SafetyVerletzung sind im Gegensatz zu den Auswirkungen einer ReliabilityVerletzung katastrophal. Das bedeutet die Fehler kosten unterschiedlich viel Geld.

Um Safety bzw. Reliability zu erreichen werden unterschiedliche Methoden des Systemdesigns angewandt. Oft stehen die Ziele in **Konflikt** miteinander (z.B. Zugsignale: alle Ampeln auf rot ist sicher, aber kein Zug kann fahren, wodurch das Zugsystem nicht funktionsfähig ist).

Wenn es keinen sicheren, nichtfunktionalen Systemzustand gibt (d.h. das System ist **fail operational**), sind Safety und Reliability jedoch eng verwandt (z.B. FlybyWire). Denn nach dem Start gibt es keinen safe (nicht-funktionierenden) system state.

2.4. Reliability vs Availability

7.2013, Ausarb. 2009

Availability ist als Kenngröße nur dann relevant, wenn die Möglichkeit der Wartung existiert.

Beispiele

Bei einem MarsRover muss die Missions-Reliability möglichst hoch sein, da das System ab dem ersten Ausfall für immer unbrauchbar wird.

Ein System zur automatisierten Montage muss jedoch eine gewisse Anzahl an Stücken pro Jahr produzieren, die Availability ist daher der wichtigste Parameter, denn dieses System kann notfalls zur Wartung auch abgeschaltet werden.

2.5. Wie hoch ist die Availability in einem System, das nicht gewartet wird?

7.2013, Ausarb. 2009

Ist ein System ab dem ersten Ausfall nie wieder einsatzfähig, beträgt die Availability Null:

$$A = \frac{\text{Zeitdauer der Einsatzfähigkeit}}{\text{Zeitdauer der Einsatzfähigkeit} + \text{Zeitdauer der nicht Einsatzfähigkeit}} = \frac{< \infty}{\infty + (< \infty)} = \frac{< \infty}{\infty} = 0$$

$$\frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

2.6. Spezifikation

Ausarb. 2009

Alle Attribute der Dependability basieren auf einer Spezifikation. Die Spezifikation und die Analyse des möglichen Verhaltens eines Systems und dessen Auswirkungen sind die schwierigsten Aufgaben beim Design eines dependable Systems.

Eine gute Spezifikation ist:

- exakt
- konsistent
- vollständig
- verbindlich (authoritative)

Oft werden Spezifikationen auf unterschiedlichen Ebenen verwendet, z.B. funktionelle, Reliability, SafetyEbene

3. Kapitel (Fault-tolerance and Modeling)

3.1. Warum werden beim Modellieren von zuverlässigen Systemen gerne konstante Fehlerraten angenommen?

2010

Die **Fehler-Dichtefunktion** $f(t)$ zum Zeitpunkt t ist die Anzahl der Fehler im Zeitabschnitt Δt :

$$f(t) = \frac{dQ(t)}{dt} = -\frac{dR(t)}{dt}$$

wobei hier $Q(t)$ die Fehlerwahrscheinlichkeit angibt. Die **Fehlerrate** $\lambda(t)$ ist definiert als die Anzahl der Fehler im Zeitabschnitt Δt in Bezug auf die Anzahl der korrekten Komponenten zum Zeitpunkt t :

$$\lambda(t) = \frac{f(t)}{R(t)} = -\frac{dR(t)}{dt} \frac{1}{R(t)}$$

Im Fall von konstanten Fehlerraten ($\lambda(t) = \lambda$) ist es so dass die Reliability definiert werden kann, als $R(t) = e^{-\lambda t}$.

$$f(t) = \frac{dR(t)}{dt} = -\lambda e^{-\lambda t}$$

Das bedeutet, es ist viel einfacher zu berechnen durch eine konstante Fehlerrate. Außerdem können die Fehlerraten aus bekannten Bauteilen aus Tabellen ausgelsen werden und direkt in der Formel verwendet werden. Außerdem werden von einigen Modellen (z.B. Markov-Modellen) nur konstante Fehlerraten unterstützt.

Wenn die Fehlerraten mit der Zeit zu oder abnehmen, kann auch die **Weibull verteilte Fehlerrate** verwendet werden: Dabei wird die Variable α eingeführt:

1. $\alpha < 1$ Fehlerrate fällt mit der Zeit
2. $\alpha = 1$ konstante Fehlerrate
3. $\alpha > 1$ Fehlerrate steigt mit der Zeit

In diesem Fall ist λ definiert, als ($\lambda(t) = \alpha\lambda(\lambda t)^{\alpha-1}$) und somit gilt für $R(t) = e^{-(\lambda t)^\alpha}$.

$$f(t) = \frac{dR(t)}{dt} = -\alpha\lambda(\lambda t)^{\alpha-1} e^{-(\lambda t)^\alpha}$$

In Halbleiteranwendungen wird häufig die **Lognormal-verteilte Fehlerrate** verwendet. Diese passt sich gut an die Halbleiterlebensdauer an.

3.2. Wahrscheinlichkeitstheoretische Modellierung was gehört dazu?

Simple Blockdiagram, Arbitrary⁷ Blockdiagram, Markov Model, General Stochastic

⁷arbitrary: frei, beliebig, willkürlich

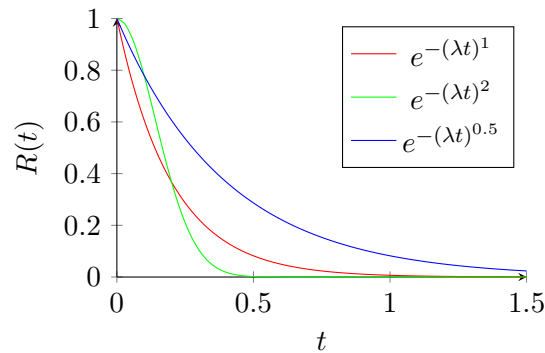


Abbildung 1: Weibull verteilte Fehlerrate

3.3. Simple/ Arbitrary Block Diagram, Markov Model, General Stochastic Petri Net?

7.2013, Ausarb. 2009

Alle diese Modelle gehören zur strukturellen wahrscheinlichkeitstheoretischen Modellierung. Zur vereinfachung werden die folgenden Annahmen gemacht: die Teilkomponenten sind unterscheidbar und jede Komponente hat eine eigene Fehlerrate. Modell basiert auf den Verbindungen der einzelnen Komponenten.

3.3.1. Simple Block Diagram

Modellierung beliebiger Kombinationen von seriell und parallel verbundenen Komponenten:

Für serielle Verbindungen gilt:

die Reliability nimmt ab, je mehr Komponenten seriell geschaltet werden:

$$R_{\text{seriell}}(t) = \prod_{i=1}^n R_i(t)$$

$$Q_{\text{seriell}}(t) = 1 - R_{\text{seriell}} = 1 - \prod_{i=1}^n R_i(t) = 1 - \prod_{i=1}^n (1 - Q_i(t))$$

Für parallele Verbindungen gilt:

die Reliability nimmt zu, je mehr Komponenten parallel geschaltet werden:

$$Q_{\text{parallel}}(t) = \prod_{i=1}^n Q_i(t)$$

$$R_{\text{parallel}}(t) = 1 - Q_{\text{parallel}} = 1 - \prod_{i=1}^n Q_i(t) = 1 - \prod_{i=1}^n (1 - R_i(t))$$

Vorteil:

- leichte Berechnung für konstante FailureRaten

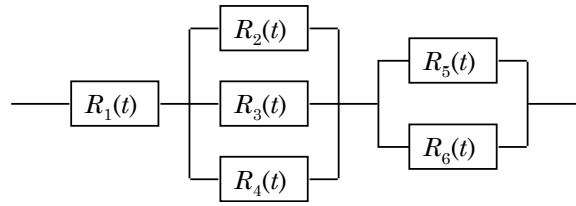


Abbildung 2: Simple Blockdiagramm

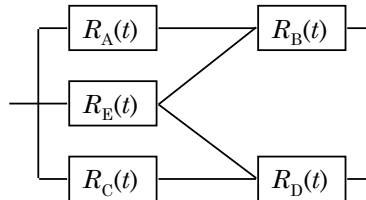


Abbildung 3: Arbitrary Block Diagramm

Nachteile:

- implizite Annahme der Unabhängigkeit von Fehlern
- Wartung kann nicht modelliert werden
- es können nur parallele oder serielle Verbindungen modelliert werden
- es kann nur aktive Redundanz unter der Annahme von FailSilence modelliert werden

3.3.2. Arbitrary Block Diagram

Wie Simple Block Diagram, aber die Beschränkung auf serielle und parallele Verbindungen fällt weg.

Hierbei wird bei der Berechnung der Reliability das Inklusions-Exklusionsprinzip angewandt:

$$R_{block}(t) = R_{AB} + R_{BE} + R_{DE} + R_{CD} - R_{ABE} - R_{ABCD} - R_{BDE} - R_{CDE} + R_{ABCDE}$$

Wobei beispielsweise R_{ABC} definiert ist als:

$$R_{ABC} = R_{seriell}(A, B, C)$$

3.3.3. Markov Modell**Geeignet zum Modellieren von**

- Allgemeinen Strukturen (aktive und passive Redundanz, VotingRedundanz)
- Systemen mit komplexen Abhängigkeiten (Fehler müssen nicht als unabhängig angenommen werden)

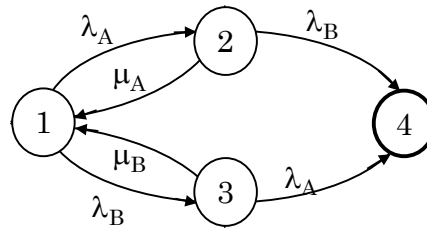


Abbildung 4: Markov-Modell von 2 Parallelen Komponenten

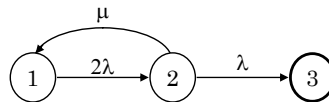


Abbildung 5: Übergänge identischer Komponenten zusammenfassen

- Coverage Effects

Einschränkungen

- Markov Property: Das Verhalten des Systems ist unabhängig von seiner Geschichte (abgesehen vom vorhergehenden Zustand)
- nur konstante FailureRates

Design

Identifikation von relevanten Zuständen, Übergänge zwischen den Zuständen erkennen und anschließende Konstruktion des Markovgraphen.

Evaluierung

Aufstellen der Differentialgleichungen, anschließend lösen (z.B. mit SHARPE)

Die Integration der Differentialgleichungen ergibt die MTTF.

z.B: die Differentialgleichungen aus Abbildung 5:

$$\begin{aligned}\frac{dP_1(t)}{dt} &= -2\lambda P_1(t) + \mu P_2(t) \\ \frac{dP_2(t)}{dt} &= -2\lambda P_1(t) - (\lambda + \mu)P_2(t) \\ \frac{dP_3(t)}{dt} &= +\lambda P_2(t)\end{aligned}$$

Die MTTF wäre dann wie folgt:

$$MTTF = \int_{t=0}^{\infty} (P_1(t) + P_2(t)) dt = T_1 + T_2$$

Mit den folgenden Grenzwahrscheinlichkeiten:

- $P_1(0) = 1$

- $P_2(0) = 0$
- $P_3(0) = 0$
- $P_1(\infty) = 0$
- $P_2(\infty) = 0$
- $P_3(\infty) = 1$

3.3.4. GSPN General Stochastic Petri Net

Erweiterung von basic Petri Nets: Übergangsdelayes können deterministisch gleich Null oder Zufallsvariablen unterschiedlichster Verteilungen (z.B. exponentiell) sein.

GSPNs sind isomorph zu kontinuierlichen MarkovKetten (d.h. alles was mit MarkovKetten modelliert werden kann, kann auch mit GSPNs modelliert werden und umgekehrt), allerdings bieten sie wesentlich mehr Mechanismen, sodass allgemeine Systeme, Algorithmen und Prozesse leichter modelliert werden können. Im Gegensatz dazu werden MarkovKetten für große Systeme schnell sehr komplex.

Die Konvertierung von GSPNs in MarkovKetten sowie deren Auswertung kann komplett automatisiert erfolgen.

Die folgenden Schritte müssen durchgeführt werden:

- Modellkonstruktion (bottom-up oder top-down)
- Modellvalidierung (strukturelle analyse, formale Beweise)
- Definition von Markings und Transition-Fireings.
- Konvertierung zur Markovkette (kann automatisiert erfolgen)
- Lösung der Markovkette

3.4. Markoveigenschaft

Ausarb. 2009

Das Verhalten des Systems ist zu jedem Zeitpunkt unabhängig von der Vorgeschichte, abgesehen vom letzten Zustand.

3.5. Modellierung für Software?

2010

Reliability Growth Model!!!!

Reliability Growth Model!!!!

3.6. Reliability Growth Model?

Ausarb. 2009

Probabilistic Structural Modeling ist für Software nicht gut geeignet, da es bei Software keine ausreichend definierten individuellen Komponenten gibt, die Komplexität sehr hoch ist, keine **Unabhängigkeit der Fehler** angenommen werden kann (da nur eine CPU und ein Speicherbereich für verschiedene Prozesse existiert), Aspekte der Echtzeitverarbeitung nicht modelliert werden können und Parallelität und Synchronisierung nicht beachtet werden (außer bei GSPNs)

Dafür sind aber Reliability Growth Models besser geeignet:

diese basieren auf der Idee einer Iterativen Verbesserung nach dem Schema testing → correction → retesting

Dabei werden keine Annahmen über die Systemstruktur (insbesondere über die Aufteilung in Komponenten) getroffen.

Z.B. Software reliability growth: SoftwareTest → Aufzeichnen der Zeitperioden zwischen aufeinander folgenden Fehlern → ausbessern der Fehler Ausgehend von den gemessenen Zeiträumen (Perioden) kann auf die zukünftigen Perioden zwischen zwei Fehlern geschlossen werden (und z.B. als MTTF angegeben werden)

Ziele:

- Disziplinierte und Kontrollierte Verbesserung der Reliability
- Extrapolieren von Zukünftiger Reliability auf Basis der Aktuellen Reliability
- Abschätzen des Aufwands für Test, Verbesserung und ReTest

Vorteile:

- Erlaubt Modellierung von alterungsbedingten Fehlern und DesignFehlern
- Kann für Hardware und für Software eingesetzt werden

Nachteile:

- Die Zuverlässigkeit dieses Modells variiert stark

Beim Testen der Software werden die Zeiten zwischen den Fehlern t_1, t_2, \dots, t_i gespeichert und von diesen Zeiten abhängig wird die Vorhersage von $T_i = \text{MTTF}$ (und natürlich auch T_{i+1}, T_{i+2}, \dots) gemacht.

3.7. Assumption⁸ Coverage?

7.2013, Ausarb. 2009

Jedes Modell/Design basiert auf einer Menge von Annahmen über das Verhalten der Komponenten in der Umgebung. Die Assumption Coverage ist die Wahrscheinlichkeit, dass die Annahmen dem Szenario der realen Welt entsprechen.

Die berechnete Reliability eines Systems muss immer noch mit der Assumption Coverage multipliziert werden. Dadurch kann die Reliability oft empfindlich reduziert werden. Ein System mit

⁸assumption: Annahme, These, Vermutung

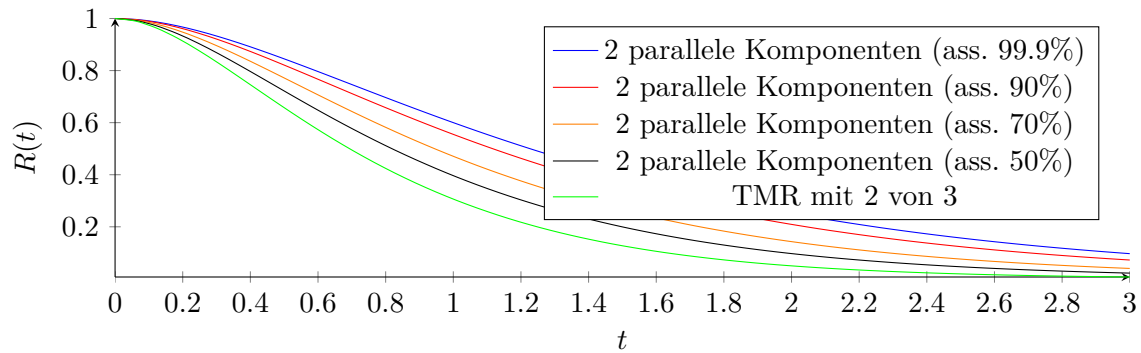
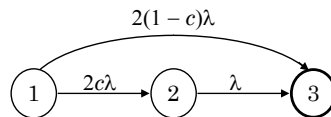


Abbildung 6: Assumption Coverage

Abbildung 7: 2ActiveRedundant System, Assumption Coverage c , Failure Rate λ

perfekter Reliability ist immer nur so gut wie die Assumption Coverage, die der Berechnung der Reliability zugrunde liegt.

Beispiel: 2AktiverRedundanz (zwei parallel geschaltete Komponenten) vs. **TMR** (Triple Modular Redundancy): Reliability von 2AktiverRedundanz höher, beruht jedoch auf der Annahme, dass sich die Komponenten **failsilent** verhalten. TMR beruht auf der Annahme, dass sich Komponenten **failconsistent** verhalten, was einer AssumptionCoverage von annähernd 1 entspricht (byzantinische Fehler sind sehr selten).

Das bedeutet, die assumption-coverage von einem active-redundant system muss immer kleiner als 1 sein.

3.8. Was ist Coverage allgemein und was bedeutet sie beim MarkovGraphen?

Ausarb. 2009

Unter der Abdeckung (Coverage) eines Fehlermodells versteht man die Anzahl von Fehlern, die im Modell beschreibbar sind, zu der Anzahl von tatsächlich auftretenden Fehlern.

Assumption Coverage

Zu welchem Prozentsatz stimmen die getroffenen Annahmen mit der Realität überein? (siehe Abbildung 7)

Error Detection Coverage

(bei aktiver Redundanz) Wie viel Prozent der Fehler einer Komponente werden von einem Switch erkannt, so dass dieser zur funktionierenden Komponente umschalten kann.

Beim Markov- Graph muss die Coverage entsprechend berücksichtigt werden. D.h. die Wahrscheinlichkeit das der Fehler innerhalb der fault- semantic liegt muss bei den "normalen"Übergängen berücksichtigen. Mit der Gegenwahrscheinlichkeit führt dann zusätzlich noch eine Kante direkt zum Fehlzustand.

4. Kapitel (Processes and Certification Standards)

4.1. Probability/Severity

Ausarb. 2009

Das primäre Ziel von Safety ist eine umgekehrt proportionale Beziehung zwischen Wahrscheinlichkeit und Schwere eines Fehlers herzustellen: Wahrscheinliche Fehler sollen keine schwerwiegenden Konsequenzen haben, während sehr sehr selten Fehler mit katastrophalen Auswirkungen auftreten dürfen.

Klassen der Severity:

- No Safety Effect
- Minor
- Major
- Hazardous (starke Reduktion der Sicherheit, zu Schaden kommen einer relativ kleinen Gruppe von Personen)
- Catastrophic (Absturz des Flugzeugs)

Klassen der Probability:

- Probable: ein oder mehrere Vorkommen im Leben eines Flugzeugs, $p > 10^{-5}$ pro Flugstunde
- Improbable: Fehler tritt wahrscheinlich nicht im Leben eines Flugzeugs auf, könnte aber bei einigen wenigen Flugzeugen eines Typs auftreten, $10^{-9} < p < 10^{-5}$
- Extremely Improbable: Fehler tritt wahrscheinlich kein einziges mal in irgendeinem Flugzeug des selben Typs auf, $p < 10^{-9}$

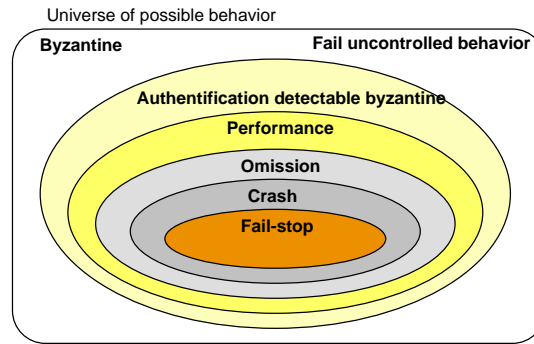


Abbildung 8: Hierarchy of failure modes

5. Kapitel (Failure modes and models)

5.1. Failure Modes?

7.2013, Ausarb. 2009

Fehler sind definiert nach dem Verhalten, das an der Systemschnittstelle beobachtet werden kann.

Hierarchische Klassifikation:

Byzantinische, Arbitrary (beliebige) Failures keine Einschränkung des Verhaltens, z.B. kann das Fehlverhalten von zwei unterschiedlichen (korrekt arbeitenden) Komponenten unterschiedlich wahrgenommen werden.

Authentication detectable byzantine Failures wie Byzantinisch, jedoch können Nachrichten von anderen Systemen nicht gefälscht werden (nur sinnvoll bei verteilten Systemen)

Performance Failures: Resultate sind korrekt bezogen auf den Wert, jedoch treffen sie zu früh oder zu spät ein (early oder late failures)

Omission⁹ Failures: Resultate kommen entweder rechtzeitig oder nie an (Spezialfall von Performance Failures)

Crash Failures: Bei Auftreten eines Fehlers sendet die Komponente in der Folge keine Resultate mehr. (Spezialfall von Omission Failures)

FailStop Failures: Spezialfall von Crash Failures, der von anderen Systemen erkannt werden kann. Außerdem kann der letzte gültige Zustand von einem stabilen Datenspeicher gelesen werden.

Klassifikation nach Auftreten

- Value Failures (byzantinisch, authentication detectable byzantinisch)
- Timing Failures (performance, omission, crash, failstop)

Klassifikation nach Wahrnehmung des Users

- Consistent Failures: von allen Usern gleich wahrgenommen (performance, omission, crash, failstop)

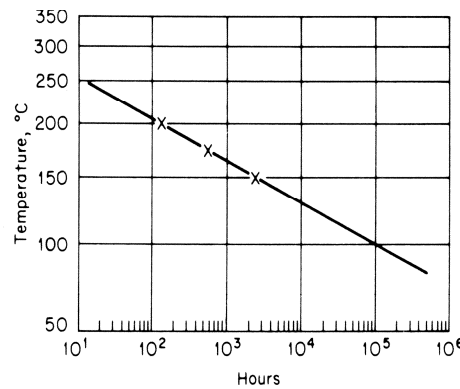


Abbildung 9: Arrhenius equation

- Inconsistent Failures: von unterschiedlichen Usern unterschiedlich wahrgenommen (byzantinisch, a.d. byzantinisch)

Klassifikation nach Auswirkungen auf die Umgebung

- Benign¹⁰ Failures: Höhe des Schadens im Fehlerfall ungefähr gleich des Nutzens des funktionsfähigen Systems
- Catastrophic failures: Schaden im Fehlerfall wesentlich Höher als Nutzen des funktionsfähigen Systems, insbesondere Schaden für menschliche Gesundheit/ menschliches Leben.

5.2. Arrhenius equation

Ausarb. 2009

Gibt den Zusammenhang zwischen Aktivierungsrate von Fehlern und Temperatur von Halbleiterbauteilen an:

$$R = R_0 \cdot e^{\frac{E_A}{kT}}$$

Wobei R_0 konstant ist, T die absolute temperatur (K) angibt, E_A die aktivierungsenergie in eV angibt und k die Boltzmannkonstante mit $8.6 \cdot 10^{-5} eV/K$ angibt.

5.3. Accelerated Stress Testing?

Wird verwendet um die InfantMortalityFailures schon vor der Auslieferung zu erkennen, und um die zu erwartende Fehlerrate mit relativ kurzen Testperioden abschätzen zu können. Möglichkeiten des künstlichen Beschleunigens von Fehlerauftrittsintervallen:

- Erhöhung der Temperatur
- Verringerung der Temperatur
- schnelles Abwechseln von hohen und niedrigen Temperaturen
- Erhöhung der Spannung

¹⁰benign: gutartig

- Erhöhung der Temperatur und der Spannung
- Erhöhung der Temperatur, Spannung und Feuchtigkeit
- Beschießen mit AlphaPartikeln

Für Software nicht anwendbar, da nicht bekannt ist, wie man „Stress“ für Software charakterisieren und simulieren kann. Außerdem gibt es keine Gleichung wie die von Arrhenius, die die Aktivierung von Fehlern beschreibt.

5.4. Safety-Analysis

7.2013, Ausarb. 2009

Die SafetyAnalyse beschäftigt sich mit dem gesamten Lebenszyklus eines Projekts, von der Spezifikation bis zur Modifikation. Wichtig sind Definitionen von Zuständigkeiten, Kommunikation mit anderen Gruppen, vollständige Dokumentation, Analyse komplexer Prozesse und ManagementProzeduren (Meetings, TimeScheduling, etc.).

Hauptfragen der SafetyAnalysis sind:

- welche Hazards sind möglich (**Hazard Analysis**)
- wie kann es zu diesen Hazards kommen (**Accident Sequencing**)
- wie wahrscheinlich kommt es zu Hazards (**Quantitative Analyse**)

Safety Analysis Methodologies:

Preliminary Hazard Analysis (PHA) Stellt die erste Phase in jeder SafetyAnalyse dar. System Hazards, kritische Zustände und FailureModes werden definiert, kritische Elemente identifiziert und Konsequenzen und Wahrscheinlichkeiten der hazardous Events bestimmt und festgelegt. Die gewonnenen Erkenntnisse sind Thema genauerer Analyse in darauf folgenden Phasen.

Hazards and Operability Study (HAZOP) Systematisches Suchen nach Abweichungen, die zu Hazards führen könnten. Dazu wird für jeden Teil des Systems eine Spezifikation der „Intention“ erstellt, danach kann das System nach Abweichungen durchsucht werden. Es werden Guide Words anhand einer CheckListe abgearbeitet, um verschiedene Typen von Abweichungen zu spezifizieren: NO, NOT, MORE, LESS, AS WELL AS, PART OF, REVERSE, OTHER THAN

Die Analyse wird von einem Team bestehend aus unterschiedlichen Spezialisten durchgeführt.

Diese Methode ist gut um die Kommunikation zwischen beteiligten Personen zu erhöhen und um Ausgangspunkte für detaillierte Studien zu schaffen, jedoch ist sie nicht gut Standardisiert.

Action Error Analysis (AEA) Beschäftigt sich mit den potentiellen Fehlern, die Menschen bei der Verwendung, Wartung, Kontrolle und Überwachung machen können.

Die einzelnen Schritte in Bedienungsabläufen werden aufgelistet, für jeden Schritt eine Liste möglicher Fehler sowie deren Ursache (gewisse Aktionen nicht durchgeführt, in falscher Reihenfolge durchgeführt, etc..) erstellt. Darauf aufbauend kann analysiert werden, wie man Kontrolle über diese Ursachen bekommen kann.

AEA beschäftigt sich nicht mit den Gründen für Bedienungsfehler.

AEA ist besonders für Software im Bereich des User Interface Design wichtig.

Fault Tree Analysis (FTA) Graphische Repräsentation von Kombinationen, die zu Hazards führen können. Dabei wird vom Hazard als Wurzel des Baums ausgegangen, die Verzweigungen finden an ODER oder UND Symbolen statt. Die einzelnen Knoten sind Events.

Kann als quantitative Methode verwendet werden.

Einschränkungen: es wird angenommen, dass Fehler binär sind, also eine Komponente entweder komplett oder garnicht ausfällt. Bäume werden schnell sehr groß und unverständlich, und sind nicht mathematisch eindeutig.

Event Tree Analysis (ETA) Faults werden als Events dargestellt, deren mögliche Konsequenzen ermittelt werden. Umgekehrte Vorgehensweise wie bei FTA: es wird mit den Events begonnen.

Kann als quantitative Methode verwendet werden.

Einschränkungen: Detaillierte Analysen können wegen des Großen Umfangs nicht erstellt werden, parallele Sequenzen, unvollständige, zu frühe oder zu späte Aktionen werden als Fehlerursache nicht in Betracht gezogen.

Failure Modes and Effect Analysis (FMEA) Der Designer muss systematisch die Fragen „Wie kann eine Komponente ausfallen?“ und „Was passiert in diesem Fall?“ beantworten. Dazu wird das System in Komponenten zerlegt (→ Block Diagram), für die die Failure Modes, die Ursache und die Bedeutung von Fehlern identifiziert werden. Es wird ermittelt, wie Failures erkannt werden können und wenn notwendig Empfehlungen zu geeigneten Kontrollmaßnahmen gegeben.

Die Analyse wird durch standardisierte Tabellen unterstützt.

Einschränkungen: Kombinationen von technischen und menschlichen Fehlern werden oft nicht bedacht und es ist schwierig gleichzeitige Fehler zu Analysieren.

Failure Modes, Effect and Criticality Analysis (FMECA) Wie FMEA, jedoch wird besonderes Augenmerk auf Criticality Aspekte gelegt. (Criticality: Wie viel Spielraum besteht zwischen korrektem Verhalten laut Spezifikation und Hazards)

CauseConsequence Analysis Kombination aus FTA und ETA: Startpunkte der Analyse sind kritische Events, ETA wird Verwendet um die Konsequenzen, FTA um die Ursachen zu ermitteln. Diese Methode ist sehr flexibel und gut dokumentiert, hat jedoch viele Nachteile der FTA (Baum wird schnell zu groß und unübersichtlich, bestimmte Kombinationen werden nicht bedacht, etc.)

6. Kapitel (System aspects of dependable computers)

6.1. Maintenance (warum besser als FaultTolerance)?

Ausarb. 2009

Maintenance macht ein System zwar komplexer, aber in wesentlich geringerem Ausmaß als FaultTolerance. Dadurch sinkt auch die Wahrscheinlichkeit von **DesignFaults**. Wannimmer es möglich ist, sollte Maintenance der Fehlertoleranz vorgezogen werden.

Es benötigt Fehlererkennung und Fehlerbeschränkung, aber dafür keine Recovery oder Fehlerbehandlung.

ErrorDetection wird benötigt, jedoch kein **FaultTreatment** auf Systemebene

Limitierungen

Allerdings ist Maintenance als Mittel zu Dependability nur möglich, falls System-Down-Times erlaubt sind (d.h. es handelt sich um ein **FailStop** oder **FailSafeSystem** wie z.B. Zugsignale, nicht jedoch um ein **FailOperationalSystem** wie z.B. Flugzeug). Außerdem kann nur begrenzte Reliability erzielt werden. **Preventive Maintenance** ist nur dann sinnvoll möglich, wenn die auswechselbaren Komponenten konstante oder steigende FailureRates haben und die **InfantMortality** (Kinderkrankheiten) gut kontrolliert werden kann.

Maintenance Schritte:

- Fehlererkennung
- Wartungsaufruf
- Wartungspersonal trifft ein
- Diagnose
- Vorrat von Ersatzteilen
- defekte Komponenten tauschen
- Systemtest
- Systemreinitialisierung
- Synchronisation mit der Umgebung

Aspekte

Wartungskosten immer mit systemkosten vergleichen, gewünschtes Wartungspersonal (Nummer, Ausbildung, Werkzeug, Wo befinden sie sich, 0-24,...)

Eigenes Ersatzteillager oder Ersatzteile erst bestellen, wenn benötigt?

Außerdem wichtig ist die Doku, geplante tests, die systemstruktur, gute fehlermeldungen, die größe und komplexität der Ersatzteile, usw.

6.2. Systematic VS app-specific fault-tolerance?

7.2013

Im Gegensatz zur systematischen Fehlertoleranz, welche versucht durch die Replikation gleichartiger Komponenten Fehler zu erkennen und zu tolerieren, baut die anwendungsspezifische Fehlertoleranz auf ein Modell der Wirklichkeit auf. Anhand dieses Modells werden bei der anwendungsspezifischen Fehlertoleranz Plausibilitätsprüfungen abgeleitet, welche Fehler erkennen sollen. Des Weiteren kann auf Basis des Modells eine Schätzung des Zustands und dadurch ein (eingeschränkter) Betrieb erfolgen.

Anwendungsspezifische Fehlertoleranz:

Der physikalische Prozess, mit dem das Computersystem interagiert unterliegt gewissen physikalischen Gesetzen und dadurch Beschränkungen. Diese Beschränkungen können im Computersystem implementiert werden um Fehler zu identifizieren. Die Plausibilitätsprüfungen sind dabei anwendungsspezifisch und ermöglichen ein "fail-stop" Verhalten des Systems. Um ein fail-operational Verhalten des Systems zu implementieren kann zusätzlich eine Abschätzung des Betriebszustandes (state-estimation) erfolgen.

Vorteile:

- forward- und backward-recovery möglich
- Keine zusätzlichen Kosten für replizierte Komponenten
- Kein Zuwachs an Komplexität auf Systemebene

Nachteile:

- Die Fehlererkennung ist durch eine Grauzone eingeschränkt
- Für manche Anwendungsgebiete existieren keine brauchbaren Plausibilitätsprüfungen
- Die Qualität des bereitgestellten Dienstes ist im Fehlerfall geringer als im Normalbetrieb
- Die korrekte Funktion des Systems hängt von der Schwere des Fehlers ab
- Zuwachs an anwendungsspezifischer Komplexität
- Anwendung und Fehlertoleranz sind stark verflochten

Systematische Fehlertoleranz

Bei der systematischen Fehlertoleranz werden keine Annahmen über den zugrundeliegenden physikalischen Prozess getroffen sondern eine etwaige Abweichung replizierter Komponenten zur Fehlererkennung eingesetzt. Replika müssen daher im Normalfall korrespondierende Ergebnisse liefern um Abweichungen im Fehlerfall erkennen zu können (replica-determinism). Diese Eigenschaft muss durch ein agreement protocol sichergestellt werden.

Bei der systematischen Fehlertoleranz wird davon ausgegangen, dass manche, aber nicht alle Komponenten fehlerhaft sind. (n-resiliency) Ein fail-stop Verhalten kann hier durch die Divergenz von Ergebnissen erreicht werden, ein fail-operational Verhalten durch mehrfach redundante Auslegung von Subsystemen.

Vorteile:

- Kein Anwendungswissen erforderlich
- Keine state-estimation erforderlich
- Unabhängige Bereiche der Applikation
- Qualität des bereitgestellten Dienstes ist unabhängig von Fehlern
- Keine Steigerung der anwendungsspezifischen Komplexität
- Fehlertoleranz kann transparent erfolgen

Nachteile:

- Benötigt Replica-Determinismus
- Die korrekte Funktion des Systems hängt von der Anzahl der eingesetzten Komponenten ab
- Nur backward-recovery
- Zusätzliche Kosten für replizierte Komponenten
- Steigerung der Komplexität auf der Systemebene

6.3. Systematic VS app-specific fault-tolerance?

7.2013

Systematisch

Replizieren der Komponenten, Abweichung der Komponenten zur Fehlererkennung, redundante Komponenten für kontinuierlichen Service

Application Spezifisch

vernünftige Checks für die Fehlererkennung (basierend auf einem realen Modell), außerdem die Einschätzung für kontinuierliche Services

Vergleich

Bei den Systematischen Fehlertoleranzsystemen wird mit replizierung gearbeitet, während die Applikationsspezifischen Fehlertoleranzsysteme keine replizierung benötigen. Im Fehlerfall können so bei systematischen Systemen durch Abweichungen die Fehler erkannt werden. Plausibilitätschecks sind somit nur für Applikationsspezifische Systeme erforderlich. Bei diesen Systemen ist auch Kenntnis der Anwendung erforderlich, bei Systematischen systemen nicht.

Bei einem systematischen System kann exakt zwischen einem Fehler und korrektem Verhalten unterschieden werden. Bei Applikationsspezifischer Fehlertoleranz nicht. (Graue Zone)

Außerdem sind bei der systematischen Fehlertoleranz keine Zustandsabschätzungen notwendig das system ist außerdem unabhängig von den Anwendungsgebieten. Bei Applikationsspezifischer Fehlertoleranz gibt es manchmal fehlende oder mangelnde Plausibilitätschecks für manche Anwendungsgebiete. Außerdem ist die Qualität der Zustandsschätzungen kleiner, als die bei normaler Anwendung.

die Servicequalität im fehlerfall bei systematischen systemen ist jedoch mit fehlerhaften replizierten Komponenten unabhängig davon, ob sie fehlerhaft sind, oder nicht. Das korrekte system hängt ab von der anzahl der korrekt funktionierenden Replikationen und deren fehlersemantik.

Beide Systeme bieten Backward recovery, aber das Application Specific system bietet auch forward recovery. Auch ist das Application Specific system günstiger, was die anzahl der komponenten angeht, denn hier fallen keine kosten für die redundanz an.

Bei der systematischen Fehlertoleranz kommt es zu einer Trennung von fehlertoleranz und applikationsfunktionalität. Außerdem kann die Fehlertoleranz von der applikation transparent behandelt werden. Bei der Applikationsspezifischen Variante sind beide tief ineinander verflochten.

Beide zusammen

in der Praxis werden meist beide arten verwendet, da die sicherheit und die Kosten einen Guten Kompromiss erfordern.

Außerdem wäre komplexe Software ohne Fehlertoleranz nicht beherrschbar. Systematische fehlertoleranz erlaubt dabei das Testen und Validieren der Mechanismen unabhängig von der software.

6.4. Replica Determinismus?

7.2013, Ausarb. 2009

Definition:

„Korrekte Replikate erzeugen korrespondierende ServiceOutputs und/oder ServiceStates unter der Annahme, dass alle Server der Gruppe im gleichen Zustand starten und korrespondierende ServiceRequests innerhalb eines gegebenen Zeitintervalls ausführen.“

Diese Art von ReplikaDeterminismus wird **Replica Control** genannt.

Es ist notwendig, dass replizierte Komponenten eines **systematisch Faulttoleranten** Systems (Fault Tolerance wird unabhängig von Applikationspezifischem Wissen implementiert, also durch Replikation) im fehlerfreien Fall korrespondierende Werte zu korrespondierenden Zeitpunkten liefern, das Verhalten der Komponenten also konsistent ist. Dies ist notwendig, um zwischen fehlerhaftem und korrektem Verhalten unterscheiden zu können.

Es scheint, dass Computer deterministisch arbeiten, aber in wahrheit arbeiten sie großteils deterministisch.

Mögliche Ursachen für ReplikaNichtdeterminismus:

- unterschiedliches Runden analoger Eingangsgrößen
- unterschiedliche Wahrnehmung der Reihenfolge von Ereignissen durch nicht perfekt synchronisierte lokale Uhren
- unterschiedliche Wahrnehmung der Reihenfolge unterschiedliche Message Transmission Delays
- Inkonsistente Information über die Gruppenmitgliedschaft von Replikationen
- nichtdeterministische Programmkonstrukte Kommunikation, Synchronisation, zufällige Zahlen, etc.)
- Entscheidungen, die auf lokalen Informationen beruhen (z.B. CPUload, lokale Zeit)
- Unterschiedliche TimeoutEntscheidungen durch ClockDrifts

- Dynamische SchedulingEntscheidungen
- Verzögerungen beim Übertragen von nachrichten.
- Unterschiedliche Rundung von RealZahlen auf unterschiedlichen Implementierungen dabei können unterschiedliche implementierungen zu auseinandergehenden resultaten führen, wenn die arithmetischen Berechnungen nahe der Grenzen der Äquivalenzklassen liegen.

Kann ReplikaNichtdeterminismus nicht verhindert werden, müssen **AgreementProtokolle verwendet werden**. (Auch diese können das Problem jedoch nicht vollständig lösen)

6.5. Synchrone vs Asynchrone Systeme

2010

Synchroner Prozessor

Ein Prozessor ist synchron, wenn er mindestens einen Prozessschritt während Δ real-time steps.

Kommunikation begrenzt

Die Kommunikationsverzögerung heißt verzögert, wenn eine Nachricht, die geschickt wurde nach höchstens Φ -Schritten am Ziel ankommt (wenn kein Fehler passiert).

Synchrones System

Ein System heißt synchron, wenn Prozessoren synchron sind und die Kommunikation begrenzt ist. Real-Time systeme sind per definition Synchron.

Asynchrones System

Ein System heißt asynchron, wenn die Prozessoren asynchron sind, oder die Kommunikation unbegrenzt ist.

6.6. Was ist konsistenz?

Ausarb. 2009

Konsistenz im Allgemeinen bezeichnet die Abwesenheit von Widersprüchen. Ein Zustand vor dem Eintreten eines Fehlers ist konsistent.

6.7. Was ist konsistenz bei Replica?

Ausarb. 2009

Ein Zustand in einem verteilten System wird Konsistent genannt, wenn er während der Ausführung des Systems existieren könnte. (Part 6, Seite 74) Konsistenz ist der Zustand, in dem sich alle korrekten Prozessoren einer Gruppe auf einen Wert geeinigt haben und alle Entscheidungen endgültig sind. (Vorlesungsfolien, part6)

6.8. Consensus Protocols?

7.2013, Ausarb. 2009

Wird bei **Strictly Distributed Replica Control** (alle Replikate sind Gleichwertig) benötigt, um Konsistenz unter den Replikaten zu erzeugen. Ein KonsensProtokoll muss folgende Eigenschaften aufweisen:

- **Konsistenz:** Jeder Knoten muss zur gleichen Entscheidung kommen
- **NichtTrivialität:** Der Wert, auf den sich die Knoten einigen, muss der Eingabewert eines Knoten der Gruppe oder eine Funktion der einzelnen EingabeWerte sein
- **Terminierung:** Jeder Knoten trifft seine Entscheidung in endlicher Zeit

6.9. Reliable Broadcast

Ausarb. 2009

Wird bei Strictly Central Replica Control (eines der Replikate hat eine Führungsposition) benötigt, um Konsistenz unter den Replikaten zu erzeugen. Ein Reliable Broadcast muss folgende Eigenschaften aufweisen:

- **Konsistenz:** Alle Knoten müssen den selben Wert erhalten, dieser Wert ist endgültig
- **NichtTrivialität:** Wenn der Transmitter nicht fehlerhaft ist, erhalten alle Knoten den Eingangswert des Transmitters
- **Terminierung:** Jeder Knoten trifft seine Entscheidung in endlicher Zeit.

6.10. Lässt sich in jedem System Consensus herstellen?

2010

Asynchrone systeme können keinen consensus durch einen deterministischen Algorithmus erreichen.

Da es unmöglich ist, zwischen einer späten Antwort und einem Prozessorcrash zu unterscheiden.

Es gibt aber probabilistische Protokolle mit denen ein Konsens in einer (zu erwartenden) konstanten Number von Durchgängen hergestellt werden kann. (coin- flipping)

In synchronen Systemen geht das schon, sofern die Kriterien für die fault- tolerance erfüllt sind.

6.11. Consistency Algorithm

7.2013, Ausarb. 2009

ConsistencyAlgorithmen sind geeignet, um für Konsistenz unter den funktionsfähigen Knoten zu sorgen, selbst wenn sich k Knoten byzantinisch verhalten (bei einer Anzahl von $n \geq 3k+1$ Knoten insgesamt).

Beispiel 1:

Annahmen über das Kommunikationssystem:

- Alle Nachrichten werden korrekt übertragen
- Der Empfänger weiß, wer der Absender einer Nachricht ist
- Die Abwesenheit einer Nachricht kann erkannt werden

Der Transmitter sendet seinen Wert an alle Empfänger. Diese senden den empfangenen Wert, bzw. den DefaultWert (falls sie keinen Wert vom Transmitter erhalten haben) an alle anderen Knoten. Jeder Knoten entscheidet nach der Mehrheit der empfangenen Werte.

Beispiel 2:

Es werden signed Messages verwendet, um byzantinisches Verhalten zu erkennen

6.12. Common Mode Failure? Diagnose?

Ausarb. 2009

Common Mode Failures treten dann auf, wenn die Ursachen für einen Failure von unterschiedlichen Komponenten nicht **statistisch unabhängig** sind.

Beispiel: Mehrere Knoten an einem Bus, ein Knoten der ausfällt verhält sich nicht failsilent („Babbling Idiot“) → Die Kommunikation aller Knoten wird gestört und alle Resultate werden unbrauchbar.

Mit der **Common Mode Analysis** versucht man zu beweisen, dass als unabhängig angenommene Failures wirklich statistisch unabhängig sind. Dabei werden auch die Auswirkungen von Design, Fertigung und Wartung analysiert.

6.13. Recovery Lines?

Ausarb. 2009

Eine Menge von **RecoveryPoints** bildet eine **Recovery Line** (= ein konsistenter Zustand), wenn er folgende Bedingungen erfüllt:

- Die Menge enthält genau einen RecoveryPoint für jeden Prozess.
- Keine Nachricht wird von Prozess P vor dessen RecoveryPoint empfangen, wenn sie nicht auch vor dessen RecoveryPoint gesendet wurde. (No **Orphan Message**)
- Keine Nachricht wird von Prozess P vor dessen RecoveryPoint gesendet, wenn sie nicht auch vor dessen RecoveryPoint empfangen wurde. (No **lost Message**)

6.14. Fault Tolerance by Self-Stabilization

Ausarb. 2009

Definition:

Ein System ist genau dann selbststabilisierend, wenn folgende zwei Bedingungen erfüllt werden:

Convergence: Wenn das System von einem beliebigen Zustand startet, erreicht es einen korrekten Zustand in endlicher Zeit.

Closure: Wenn sich ein System in einem korrekten Zustand befindet, betritt es nie wieder einen inkorrekten Zustand (außer ein weiterer Fehler tritt auf)

weiteres:

Selbststabilisierende Systeme müssen nicht initialisiert werden. Statt Fehler zu verhindern und zu korrigieren, konzentriert sich dieser Ansatz darauf nach einem Fehler wieder einen Konsistenten Zustand zu erreichen.

Selbststabilisierende Systeme müssen nicht initialisiert werden, und können sich von flüchtigen Fehlern wieder erholen. (z.B: der **adaptive DSD-Algorithmus** ist selbststabilisierend)

Probleme:

Eine globale Eigenschaft (korrekter Systemzustand) kann mit ausschließlich lokalen Informationen nicht oder nur schwer hergestellt werden und es ist keine Theorie bekannt, wie selbststabilisierende Algorithmen erstellt werden und wie Rechtzeitigkeit garantiert werden kann.

6.15. Wie viele Replikas um Crash/Performance/Byzantine Failure zu tolerieren (+ warum $3k+1$ bei byzantine und nicht $2k+1$)

7.2013

man braucht

- $3k + 1$ Komponenten, um k byzantinische Fehler zu tolerieren
- $2k + 1$ Komponenten, um k PerformanceFehler zu tolerieren
- $k + 1$ Komponenten, um k Crash oder OmissionFehler zu tolerieren

Für Performance und byzantinische Fehler werden $k+1$ identische Resultate benötigt

6.16. Fail-Silent/Fail Consistent

Ausarb. 2009

FailSilent: Eine fehlerhafte Komponente erzeugt keine Ausgabe

FailConsistent: Eine fehlerhafte Komponente erzeugt falsche Ausgaben, diese werden von allen Komponenten gleich wahrgenommen.

Byzantinisch: Eine fehlerhafte Komponente erzeugt falsche Ausgaben, diese werden von allen Komponenten unterschiedlich wahrgenommen.

6.17. Single Point of Failure

Ausarb. 2009

Problem von shared Resources: fällt die gemeinsam genutzte Ressource aus, betrifft das alle Komponenten, die diese Ressource verwenden.

A. Zusatzinfos

A.1. Kapitel:

A.1.1. Gründe für niedriges Vertrauen

System on a Chip (alle Komponenten auf einem Chip), ein Chip für alles.

Interface design Komplexe Systeme \Rightarrow einfaches interface, keine verwirrungen

Operator das system benötigt interaktion von Operator.

Documentation das System ist gut dokumentiert. (Fehlerquelle wenn schlechte oder keine doku)

Komplexität ist immer schwer handhabbar.

systemstruktur ungünstige Systemstruktur (unpassende komponenten) kann zu wenig vertrauen führen.

low dependability of components das system ist so stark wie das schwächste Glied.

Kein System ist 100%-ig sicher.

A.2. Kapitel:

A.2.1. Mögliche Requirements

System Lifetime, difference or no difference between reliability and safety, warranty, environmental conditions, vibrations, power supply, ...

Stresstests

Function Test, Heat Test, Cold Test, Heat storage, Temperature shock, ...

A.2.2. Beeinträchtigung der Zuverlässigkeit(Dependability):

Ausfall (failure)

Die Einhaltung der Servicespezifikation wird erschwert. (Übergang von korrekten zu inkorrektem Service)

Fehlerzustand (error)

Teil eines Systemzustands, der evtl. zu einem Fehler führen kann.

Fehlerursache (fault)

Fehlerursache, die toleriert oder verhindert werden soll.

fault \rightarrow **error**: Fehlerursache ist aktiv, wenn diese zu einem Fehlerzustand führt.

error \rightarrow **failure**: Nicht erkannter Fehler ist latent(gebunden) \Rightarrow Fehlererkennung.

failure \rightarrow **fault**: Ausfall führt zu einer Fehlerursache eines anderen Systems, wenn das fehlerhafte system mit diesem interagiert. Fehlerzustand wird durchgereicht und beeinträchtigt den Service.

siehe Abbildung 10 auf der nächsten Seite

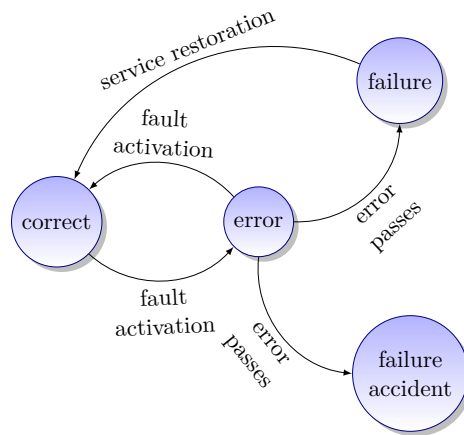


Abbildung 10: Failure-Chain

A.2.3. Fehlerklassen

Accidental, Internal, External, Physical, Human Made, Design, Operational, Permanent, transient, Intentional

A.2.4. Fehlerzustand → Ausfall?

Ob ein Fehlerzustand zu einem Ausfall führt, hängt ab von:

- von den Systemkomponenten und ob das System redundant ist.
- von der Systemaktivität
- von der Usersicht, was einen Fehler ausmacht.

A.2.5. Mittel der Systemverlässlichkeit

Fehlervermeidung (fault prevention): Methoden und Techniken zur Vermeidung von drohenden Fehlern.

Hardware: Vermeidung von Umwelteinflüssen, qualitativere Komponenten,

Software: Verwendung von OOD und OO, formale Methoden, klare Designrichtlinien

Fehlerkorrektur (fault removal) Methoden und Techniken zur Reduzierung der Anzahl und der Bedeutung der Fehler.

Verifizierung: entweder statisch (data flow, compiler checks, ...) oder dynamisch (testing, black-box, white box, ...)

Diagnose: Erkennen des Fehlers, der Verifizierung stoppte

Korrektur: Den Fehler entfernen

Fehlervorhersage (fault forecasting) : Methoden und Techniken zur Evaluierung und Modellierung der aktuellen und der zukünftigen Fehler.

Fehlerevaluierung an System ansetzen. Beispielsweise reliability, availability, maintainability, safety

Fehlertoleranz : Methoden und Techniken die Services zur Verfügung stellen, die selbst im Fehlerfall noch die Spezifikation erfüllen.

Fehlererkennung ist wichtig, um darauf reagieren zu können.

Schadensbegrenzung und -bewertung bevor auf den Fehler reagiert wird.

Fehler recovery um den Fehlerzustand zu Gunsten eines fehlerfreien Zustands zu verlassen. Hierbei gibt es 2 Arten: Backward recovery(set system back to previous error-free state and reexecute failed processing sequence) und die Forward recovery(system state is set to new error-free system state (for real-time systems)).

Fehlerbehandlung um den Fehler nicht sofort wieder zu triggern.

A.2.6. Fehlertoleranz und redundanz

Einsatzgebiet Information redundante Information, ECC, robuste Datenstrukturen

Einsatzgebiet Speicher Replikation der Komponenten (2 CPU's, Uninterruptable Power Supply)

Einsatzgebiet Zeit Replikation der Berechnungen (mehrmalige Berechnung über verschiedene oder gleiche Algorithmen, mehrmaliges Senden der gleichen Information).