SOFTWARE SAFETY ANALYSIS:

Ein hybrider Ansatz aus FTA, SDRM und ETA zur Architekturbewertung nach ISO 26262

Inhalt

Inhaltsverzeichnis

- 4.3 Modellierung externer Einflüsse: Explizite Fehlerbäume für Umweltbedingungen und Runtime-Degradation.
- 4.4 Plausibilisierung: Bewertung der resultierenden Baumstruktur (Logik-Check mit RPS-Werten).

5. Phase 5: Verhaltensanalyse mittels ETA (Event Tree Analysis)
    - 5.1 Sequenzielle Betrachtung: Analyse der zeitlichen Abfolge von Fehler, Detektion und Reaktion.
    - 5.2 Das Bow-Tie-Modell: Verknüpfung von Ursachenanalyse (FTA) und Konsequenzanalyse (ETA).
    - 5.3 Bewertung der Pfade:
        - Unterscheidung zwischen sicherem Zustand (Safe State) und Gefährdung (Hazard).
        - Darstellung der Wirksamkeit der Sicherheitsarchitektur anhand der Pfad-Verteilung.
6. Kritische Auseinandersetzung
    - 6.1 Methodische Stärken: Effizienz, Fokus auf kritische Pfade, Architekturbewertung.
    - 6.2 Grenzen und Risiken: Umgang mit semi-quantitativen Scores, Konsistenzsicherung zwischen Modellen.
    - 6.3 Abgrenzung: Vergleich mit anderen gängigen Analysemethoden.
7. Zusammenfassung

1. Motivation: Grenzen herkömmlicher Methoden

Die Entwicklung sicherheitskritischer Software im Automobilbereich, getrieben durch Standards wie die ISO 26262 (Funktionale Sicherheit), ISO 21448 (SOTIF) und zunehmend die ISO 21434 (Cybersecurity), steht vor einem fundamentalen Problem: Die etablierten Analysemethoden stammen größtenteils aus der Hardware-Welt.

Während sich mechanische Bauteile und elektronische Schaltungen physikalischen Gesetzen unterwerfen, folgt Software einer reinen Logik. Der Versuch, Hardware-Methoden unverändert auf Software zu übertragen, führt in der Praxis oft zu enormem Dokumentationsaufwand bei gleichzeitig geringem Erkenntnisgewinn für das Systemdesign. Dieses Kapitel beleuchtet die methodischen Diskrepanzen und motiviert den Bedarf für einen angepassten, hybriden Ansatz.

1.1 Systematische vs. Zufällige Fehler: Warum Hardware-Metriken nicht auf Software-Architekturen passen.

Der wichtigste Unterschied zwischen Hardware und Software liegt in der Natur des Fehlers.

Hardwarefehler sind überwiegend zufällig.
Elektronische Bauteile unterliegen Verschleiß, Alterung und physikalischem Stress (Temperatur, Vibration). Ein Widerstand kann nach 10.000 Stunden ausfallen, auch wenn das Design korrekt war. Daher ist die Verwendung von statistischen Ausfallraten (bspw. FIT – Failures In Time) und Wahrscheinlichkeitsrechnungen in der Hardware-Analyse (Hardware-Metriken nach ISO 26262 Teil 5) sinnvoll und notwendig.

Softwarefehler sind systematisch.
Software verschleißt nicht. Sie rostet nicht und altert nicht im physikalischen Sinne. Ein Fehler in der Software ist ein Designfehler, der von Anfang an im Code existiert (ein Bug). Er tritt nicht „zufällig“ auf, sondern wird unter spezifischen Eingangsbedingungen oder Zuständen deterministisch getriggert.
Daraus ergeben sich zwei Konsequenzen für die Sicherheitsanalyse:

1. **Keine echte Wahrscheinlichkeit:** Man kann für einen Software-Bug keine physikalische Eintrittswahrscheinlichkeit ($10^{-x}/h$) berechnen.
2. **Keine Redundanz durch Duplizierung:** Lässt man dieselbe fehlerhafte Software auf zwei redundanten Prozessoren laufen, werden beide exakt denselben Fehler zum exakt selben Zeitpunkt produzieren.

Klassische Ansätze versuchen oft, Software durch Pseudo-Wahrscheinlichkeiten mathematisch greifbar zu machen. Dies führt jedoch zu einer Scheingenauigkeit, die

das tatsächliche Risiko – nämlich die Komplexität der Logik und die Unvollständigkeit der Spezifikation – verschleiert.

1.2 Kritik an der isolierten SW-FMEA: Fehlender Systembezug und mangelnde Design-Relevanz.

Die Software-FMEA (Fehlermöglichkeits- und Einflussanalyse) ist in vielen Entwicklungsprozessen der Standard. Oft wird sie jedoch als rein formale Pflichtübung betrachtet („Paper Tiger“). Folgende Schwächen sind in der Praxis häufig zu beobachten, wenn die FMEA isoliert und nach Hardware-Mustern angewendet wird:

- **Verlust des Systembezugs (Bottom-Up-Problem):** Die klassische SW-FMEA arbeitet induktiv. Sie beginnt oft auf Modul- oder Unit-Ebene und analysiert Fehler Zeile für Zeile oder Funktion für Funktion. Dabei geht der Blick für das übergeordnete Sicherheitsziel (Safety Goal) verloren. Man analysiert hunderte von Detailfehlern, übersieht aber kritische Wirkungsketten, die erst durch das Zusammenspiel mehrerer Komponenten entstehen.
- **Mangelnde Design-Relevanz:** Häufig wird die SW-FMEA erst erstellt, wenn die Softwarearchitektur bereits fixiert oder der Code schon geschrieben ist. Sie dient dann als Nachweisdokumentation, treibt aber keine Designentscheidungen mehr.
- **RPN-Falle:** Die Berechnung einer Risikoprioritätszahl (RPN = S × A × E) suggeriert eine Vergleichbarkeit, die bei Software oft nicht gegeben ist. Da die Auftretenswahrscheinlichkeit (A) bei systematischen Fehlern schwer zu schätzen ist, werden oft pauschale Werte angenommen, was die Priorisierung verfälscht.
- **Fehlender Fokus:** Ohne eine vorherige Filterung neigt die SW-FMEA dazu, zu explodieren. Ingenieure verbringen Zeit damit, triviale Fehler in unkritischen Modulen zu dokumentieren, statt die komplexen Pfade in sicherheitskritischen Bereichen tiefgehend zu durchdringen.

Zusammenfassend lässt sich sagen: Eine SW-FMEA allein ist oft fleißig, aber blind für die architektonischen Schwachstellen.

1.3 Der integrierte Ansatz: Die Verknüpfung von Logik (FTA), Bewertung (SDRM) und Verhalten (ETA).

Um den spezifischen Eigenschaften von Software gerecht zu werden, benötigen wir einen Ansatz, der die Stärken verschiedener Methoden kombiniert und ihre Schwächen kompensiert. Das in diesem eBook vorgestellte Vorgehen basiert auf einer Symbiose aus drei Säulen:

1. **Die Logik (FTA):** Wir nutzen die Fehlerbaumanalyse (Fault Tree Analysis) als **deduktives** Werkzeug (Top-Down). Ausgehend vom Sicherheitsziel (und daraus dem unerwünschten Ereignis) brechen wir die Architektur logisch herunter. Dies stellt sicher, dass wir nur das analysieren, was für die Sicherheit relevant ist, und filtert irrelevante Systemteile frühzeitig aus.
2. **Die Bewertung und das Design (SDRM):** Anstelle einer klassischen FMEA nutzen wir eine **Safety Design & Rating Matrix**. Hier findet die detaillierte Bewertung der identifizierten Basis-Ereignisse statt. Wir ersetzen physikalische Wahrscheinlichkeiten durch semi-quantitative Scores (RPS – Risk Priority Score), die eine relative Priorisierung ermöglichen. Der Fokus liegt hier explizit auf dem Design von Sicherheitsmechanismen (Detektion und Reaktion).
3. **Das Verhalten (ETA):** Die Event Tree Analysis (Ereignisablaufanalyse) prüft die Wirksamkeit der Maßnahmen in der zeitlichen Abfolge. Sie beantwortet die Frage: *Führt die definierte Fehlerreaktion das System tatsächlich in einen sicheren Zustand, bevor eine Gefährdung eintritt?*

Dieser hybride Ansatz ermöglicht es, Software-Sicherheit nicht als additives „Feature“ zu betrachten, sondern als integrales Qualitätsmerkmal der Architektur zu konstruieren und nachzuweisen.

2. Phase 1: Erstellung der qualitativen Systemlogik (FTA)

Der Ausgangspunkt jeder effektiven Sicherheitsanalyse ist nicht die Frage „Wie oft geht etwas schief?“, sondern „*Warum* geht etwas schief?“. In der ersten Phase unseres Vorgehens nutzen wir die Fehlerbaumanalyse (Fault Tree Analysis, FTA), um eine logische Karte der Fehlerursachen zu erstellen. Dieser Ansatz ist streng deduktiv (Top-Down): Wir starten beim Verletzen des Sicherheitsziels und brechen dieses systematisch bis auf die Ebene der Software-Komponenten herunter.

2.1 Ableitung aus der SW-Architektur

Eine Software-FTA darf nicht im luftleeren Raum entstehen. Sie muss ein direktes Abbild der definierten **Software-Architektur** sein.

Die Erstellung der FTA erfolgt als direkte Abbildung der Software-Architektur. Der Fehlerbaum strukturiert sich nicht nach abstrakten Fehlermodi, sondern nach den **architektonischen Bausteinen** (Building Blocks) und **Datenpfaden**, die in der SW-Architektur definiert sind.

Der Prozess der FTA-Erstellung folgt dabei dem Datenpfad rückwärts:

Top Event

Das Sicherheitsziel (Safety Goal) als unerwünschtes Ereignis formuliert.

- **Beispiel:**
  1. *SSR:* „Die Software muss die Integrität von Speicher- und Buffer-Operationen sicherstellen.“
  2. *FTA Top Event:* „Verletzung der Speicher- oder Buffer-Integrität (Memory Corruption / Overflow).“

Architektonische Dekomposition (Ebene 1 & 2):

Unterhalb des Top Events wird der Baum anhand der **architektonischen Sub-Komponenten** oder **funktionalen Blöcke** aufgegliedert. Wir fragen: *„Welche architektonischen Einheiten sind an der Erfüllung dieser Anforderung beteiligt?“* Anstatt Fehlerarten aufzulisten, bilden wir die Architekturstruktur ab. Wenn eine Komponente aus drei Sub-Modulen besteht (z.B. *Input-Parser*, *Calculation-Core*, *Output-Buffer*), dann erhält der Fehlerbaum auf der nächsten Ebene exakt diese drei Äste.

- **Logik:** Da in einer seriellen Software-Architektur jeder Block funktionieren muss, um das Gesamtziel zu erreichen, werden diese Äste mit einem **ODER-Gatter** verknüpft. Ein Fehler in *einem* der Blöcke führt zur Verletzung des Top Events.
- **Die Wahl des Gatters:** Bei der Dekomposition der Architektur-Blöcke gilt folgende Regel:

- **Der Regelfall: ODER-Gatter (Serielle Abhängigkeit)**
  Software verarbeitet Daten meist sequenziell oder funktional abhängig. Wenn Modul A die Daten von Modul B benötigt, führt ein Fehler in Modul B zwangsläufig zu einem Fehler in Modul A.
    - *Logik:* Ein Fehler im Input-Parser **ODER** ein Fehler im Berechnungskern führt zum falschen Output.
    - *Anwendung:* Bei >95% der architektonischen Zerlegung.
- **Die Ausnahme: UND-Gatter (Architektonische Redundanz)**
  Ein UND-Gatter darf in Phase 1 **nur dann** gesetzt werden, wenn die Software-Architektur **explizit redundante Pfade** zur Erfüllung der *gleichen* Funktion vorsieht.
    - *Szenario:* Die Architektur definiert zwei diversitäre Algorithmen (z.B. Pfad A und Pfad B), deren Ergebnisse verglichen werden, bevor ein Output generiert wird.
    - *Logik:* Damit der Output falsch ist, müssen Algorithmus A **UND** Algorithmus B gleichzeitig versagen (oder der Vergleicher).
    - *Wichtig:* Dies betrifft echte funktionale Redundanz, noch nicht die später hinzugefügten Sicherheitsmechanismen (Monitoring), die wir erst in Phase 4 zurückführen.

Identifikation der Fehlermodi auf Unit-Ebene (Base Events)

Erst in der tiefsten Ebene, innerhalb der jeweiligen Architektur-Äste, werden die konkreten **Software-Fehler** (Base Events) identifiziert. Hierbei wird analysiert, welcher spezifische Logik- oder Datenfehler innerhalb der Unit zum Ausfall dieses Architektur-Blocks führt.

- **Fokus:** Wir betrachten Variablen, Puffer, Kontrollflüsse und Schnittstellen.
- *Beispiel im Ast ‚Output-Buffer':* „Buffer Overflow durch Index-Fehler" oder „Veraltete Daten im Puffer durch fehlenden Refresh".

Ergebnis:
Das Resultat ist eine FTA-Struktur, die deckungsgleich mit der Software-Architektur ist. Dies gewährleistet die Wartbarkeit: Ändert sich ein Modul in der Architektur, ist sofort ersichtlich, welcher Ast im Fehlerbaum angepasst werden muss.

2.2 Nutzung von LLMs: Effizienzsteigerung bei der initialen Generierung der Fehlerbäume.

Die manuelle Erstellung von Fehlerbäumen ist traditionell eine der zeitintensivsten Tätigkeiten in der Funktionalen Sicherheit. Hier greift unser Ansatz auf moderne Large Language Models (LLMs) zurück, um den Prozess zu beschleunigen.

Ein LLM (wie Mistral oder DeepSeek) versteht den semantischen Zusammenhang von Textbeschreibungen. Wir nutzen dies wie folgt:

1. **Input:** Das LLM erhält eine textuelle Beschreibung der SW-Architektur (z.B. „Modul A liest Sensor X, berechnet Wert Y und sendet an Modul B“) sowie das zu analysierende Sicherheitsziel.
2. **Prompting:** Das Modell wird instruiert, basierend auf dieser Architektur logische Fehlerpfade zu identifizieren und diese in einer strukturierten Form (z.B. textuelle Baumstruktur oder XML-Format für FTA-Tools) auszugeben.
3. **Output:** Das LLM generiert einen *Entwurf* der FTA.

Wichtig: Das LLM agiert hier als „Junior Engineer“. Es übernimmt die Schreibarbeit und die Strukturierung. Die Validierung und das „Review“ obliegen zwingend dem menschlichen Experten (Senior Safety Engineer). Die Erfahrung zeigt, dass dieser Ansatz den manuellen Aufwand der initialen Baumerstellung um ca. 70-80% reduzieren kann, sodass mehr Zeit für die inhaltliche Prüfung bleibt.

2.3 Definition der Base Events: Logische Fehlerursachen ohne Bewertung.

Die Blätter des Fehlerbaums sind die sogenannten **Base Events** (Basisereignisse). In unserer Methodik kommt ihnen eine besondere Bedeutung zu, da sie die Schnittstelle zur nächsten Phase (SDRM) bilden.

Ein Base Event in der Software-FTA beschreibt eine spezifische, logische Fehlerursache auf Ebene der Software-Einheit oder der Schnittstelle.
Beispiele für valide Base Events sind:

- *„Signal Quality Index invalid but not detected“*
- *„Calibration Data Corrupted during Load“*
- *„Calculation Overflow in Torque Module“*

In dieser Phase 1 bleiben die Base Events **rein qualitativ**. Wir weisen ihnen keine Wahrscheinlichkeiten, Scores oder FIT-Raten zu. Das Ziel ist Vollständigkeit in der Logik: Haben wir alle Pfade erfasst, die zur Verletzung des Sicherheitsziels führen können?

Erst wenn die logische Struktur des Baumes steht und vom Experten als plausibles Abbild der Architektur validiert wurde, werden die Base Events in die **Safety Design & Rating Matrix (SDRM)** exportiert, um dort bewertet zu werden.

3. Phase 2 & 3: Bewertung und Design in der SDRM (Safety Design & Rating Matrix)

Nachdem die logische Struktur der Fehlerursachen (Base Events) steht, erfolgt der Übergang von der Analyse zum Design. Anstatt eine klassische FMEA zu befüllen, nutzen wir die **Safety Design & Rating Matrix (SDRM)**. Diese Methode dient nicht primär der Dokumentation, sondern der gezielten Steuerung der Sicherheitsarchitektur durch Priorisierung.

3.1 Aufbau der Matrix

Die SDRM unterscheidet sich strukturell von einer Standard-SW-FMEA. Sie ist darauf ausgelegt, den Prozess „Problem → Bewertung → Lösung“ in einer logischen Flussrichtung abzubilden. Veraltete Metriken wie die Risikoprioritätszahl (RPN) oder Spalten für „Current Controls“ auf der Ursachenseite entfallen, um den Fokus auf aktive Sicherheitsmechanismen zu legen.

Die Struktur:

1. **Input (Links):** Import der Base Events aus der FTA (Phase 1).

ID	Item	Failure Mode	Effect	Vehicle Effect	Cause

2. **Bewertung (Mitte):** Bestimmung des **RPS Base** (Risk Priority Score) basierend auf Schwere und Auftretenswahrscheinlichkeit.

S	k_S	O	k_O	RPS1	RPS Base

3. **Filter (Trigger):** Entscheidungsschwelle für Handlungsbedarf.

Resp	FTA Ref	Base Event		SM ID	SM Description	Detection Logic		D	DC	RPS2	RPS Detection

4. **Design (Rechts):** Definition von Detektion und Reaktion sowie Berechnung des **RPS Final**.

Reaction Logic	R	R^3	RPS3		RPS Final	Validation & Verification

3.2 Bestimmung des RPS (Risk Priority Score)

Um Software-Risiken vergleichbar zu machen, ohne physikalische Ausfallraten (FIT) zu simulieren, führen wir den **Risk Priority Score (RPS)** ein. Der RPS ist ein dimensionsloser Index, der die Kritikalität eines Software-Fehlers ausdrückt.

Der **RPS Base** (Basis-Score ohne Sicherheitsmechanismus) wird durch Expertenurteil ermittelt und basiert auf zwei Faktoren:

1. **Severity (S):**
   Dies ist ein Maß für die Schwere der Auswirkung auf das Top Event bzw. das Sicherheitsziel. Der Wert wird von Experten festgelegt.
   - *Kontext:* Zwar korreliert die maximale Severity indirekt mit der ASIL-Einstufung des Sicherheitsziels, jedoch bewertet der Experte hier spezifisch, wie gravierend der jeweilige Software-Fehler zur Verletzung des Ziels beiträgt.
2. **Occurrence (O):**
   Eine qualitative Abschätzung der Auftretenswahrscheinlichkeit des Fehlers im Code.
   - *Kriterien:* Hier fließen Faktoren wie Code-Komplexität, Erfahrungswerte mit der Funktion (Legacy Code vs. Neuentwicklung) und die Stabilität der Inputs ein.

Die Skalierung erfolgt so, dass der RPS als Integer-Wert dargestellt wird (z.B. auf einer Skala bis 1000 oder logarithmisch gespreizt), um eine schnelle visuelle Erfassung zu ermöglichen.

Severity weighting				**Occurence weighting**		
					=10^(-(8-O-value))	
S-value	**significance**	**Weighting factor (k_S)**		**O-value**	**k_O**	**Reason**
10	catastrophic	1		**1**	1,00E-09	Extremely rare
9	very critical	0,9		**2**	1,00E-07	Rare
8	critical	0,5		**3**	1,00E-05	Possible
7	high	0,3		**4**	1,00E-03	Probable
6	medium	0,1		**5**	1,00E-01	Very probable
5	medium-low	0,05				
4	low	0,01				
3	very low	0,005				
2	minimal	0,001				
1	negligible	0,0001				

3.3 Filterung und Fokussierung

Der entscheidende Vorteil des RPS-Systems ist die Filterung. In der Praxis führt eine FTA oft zu einer hohen Anzahl an Base Events (z.B. 90 Events für ein komplexes Modul). Es ist weder wirtschaftlich noch technisch sinnvoll, jeden dieser Pfade mit aufwendigen Sicherheitsmechanismen zu versehen.

Das Filter-Prinzip:
Wir definieren einen **Schwellenwert** (Threshold, z.B. RPS = 100).

- **RPS Base > 100:** Der Pfad ist kritisch. Hier **muss** ein Sicherheitsmechanismus (Detektion + Reaktion) entworfen werden.
- **RPS Base ≤ 100:** Das Risiko wird als akzeptabel eingestuft (Accepted Risk). Es ist keine dedizierte Absicherung im Code notwendig.

Durch dieses Vorgehen reduziert sich die Menge der zu bearbeitenden Events signifikant (z.B. von 90 auf 30 relevante Pfade). Die Entwicklungsressourcen konzentrieren sich somit auf die architekturellen Schwachstellen.

3.4 Design der Sicherheitsmechanismen

Für die verbleibenden kritischen Pfade (RPS Base > Schwellenwert) erfolgt nun das technische Design in der SDRM. Ein Sicherheitsmechanismus in der Software besteht immer aus zwei Komponenten:

Detektion: Definition und Bewertung der Entdeckungsgüte

Wie erkennen wir den Fehler?

- *Methoden:* Range Check, Plausibilitätsprüfung, E2E-Protection, Watchdog.
- *Bewertung:* Ein Faktor (Detektionsgüte), der den RPS reduziert (z.B. Faktor 0,1 für hohe Güte).

Reaktion: Definition der Fehlerreaktion und Risikoreduktionspotenzial (R^3)

Was tun wir, wenn der Fehler erkannt wurde?

- *Grundsatz:* **„No Detection without Intention“**. Eine Detektion ohne definierte Reaktion ist wirkungslos.
- *Methoden:* Reset, Ersatzwertbildung, Safe State, Degradation.
- *Bewertung:* Einführung des Faktors R^3 **(Reaction Risk Reduction)**. Eine harte Abschaltung (Safe State) reduziert das Risiko stärker als eine bloße Fehlerspeichereintragung.

Derive Diagnostic Coverage (DC)		
D-value	**significance**	**Diagnostic Coverage (DC)**
1	Very high detection	0,99 (99%)
2	High detection	0,90 (90%)
3	Medium detection	0,60 (60%)
4	Low detection	0,10 (10%)
5	Very low detection	0,01 (1%)
6	Hardly any detection	0,001 (0,1%)
7	Virtually no detection	0,0001 (0,01%)

Zusammenhang: Abhängigkeit von Detektion und Reaktion

Der RPS Final berechnet sich aus dem Basis-Score multipliziert mit den Faktoren für Detektion und Reaktion.

$$\boldsymbol{RPS_{Final} = RPS_{Base} \times Faktor_D \times Faktor_R}$$

Das Ziel ist erreicht, wenn der $\boldsymbol{RPS_{Final}}$ unter den definierten Schwellenwert sinkt. Dies dient als Nachweis, dass die Sicherheitsarchitektur (Detektion + Reaktion) hinreichend effektiv ist.

Reaction Risk Reduction (R^3)			
R-value	**RR**	**R^3**	**significance**
1	0,99	0,01	Immediate safe state (ASIL D)
2	0,95	0,05	Planned safe state
3	0,9	0,1	Degraded mode with functional restriction
4	0,8	0,2	Severe degraded mode
5	0,7	0,3	Warning + slight degradation
6	0,5	0,5	Warning only
7	0,3	0,7	Logging with alarm
8	0,1	0,9	Logging only
9	0,01	0,99	Hardly any effect
10	0	1	No reaction

Safety Reaction Rating Guideline:
Rating 1 (Safe State, P_fail=0.01):
- Immediate transition to defined safe state
- Example: Switch off function, emergency stop
Rating 2-3 (Degraded Mode, P_fail=0.05-0.10):
- Restricted operation with reduced performance
- Example: Speed limitation, reduced sensor technology
Rating 4-6 (Warning, P_fail=0.20-0.50):
- Warning to driver/system
- Example: Acoustic/optical warning, log entry
Rating 7-9 (Logging, P_fail=0.70-0.99):
- Logging measures only
- Example: Fault memory entry, diagnosis

4. Phase 4: Validierung der Architektur und DFA (Dependent Failure Analysis)

Die Bearbeitung in der SDRM (Phase 2 & 3) liefert uns optimierte Pfade: Ein Fehler tritt auf, wird erkannt und abgefangen. Würden wir hier aufhören, hätten wir ein „schön gerechnetes“ System. In Phase 4 überführen wir diese Ergebnisse zurück in den Fehlerbaum und erweitern das Modell um die kritische Betrachtung der **Dependent Failures (DFA)**.

4.1 Rückführung in die FTA

Die in der SDRM definierten Sicherheitsmechanismen (Detektion und Reaktion) existieren nicht nur als Spalten in einer Tabelle, sie verändern die logische Struktur des Fehlerszenarios.

Strukturelles Update (Das „3-Beine-Modell“):
Jedes Base Event, das in der SDRM einen Sicherheitsmechanismus erhalten hat (RPS > Schwellenwert), wird im Fehlerbaum expandiert. Aus dem ursprünglichen Einzel-Event wird eine Sub-Struktur unter einem **UND-Gatter**.

Damit das Top-Event (Verletzung des Sicherheitsziels) über diesen Pfad eintritt, müssen drei Bedingungen gleichzeitig erfüllt sein:

1. **Das Base Event (Ursache):** Der Software-Fehler tritt auf.
2. **Versagen der Detektion:** Der Diagnose-Mechanismus (z.B. Range Check) übersieht den Fehler.
3. **Versagen der Reaktion:** Die definierte Fehlerreaktion (z.B. Safe State Transition) wird nicht korrekt ausgeführt.

Durch diese Modellierung wird der Fehlerbaum von einer reinen Ursachenanalyse zu einer **Architektur-Blaupause**. Man erkennt visuell sofort, welche Pfade „Single Point Faults“ sind (kein UND-Gatter) und welche durch Sicherheitsmechanismen geschützt sind.

4.2 Modellierung interner Abhängigkeiten - Nutzung des Beta-Faktor-Modells für Plattform-Kopplungen

Die größte Schwachstelle reiner Software-Sicherheitsanalysen ist die Annahme perfekter Unabhängigkeit. Wenn Funktion und Monitor auf derselben CPU laufen und denselben Speicher nutzen, existieren **Common Cause Failures (CCF)**, die das UND-Gatter aus Abschnitt 4.1 aushebeln.

Wir modellieren diese internen, architektonischen Kopplungen mittels des **Beta-Faktor-Modells**.

Der „Bypass“ im Fehlerbaum:
Wir fügen der Struktur ein übergeordnetes **ODER-Gatter** hinzu.

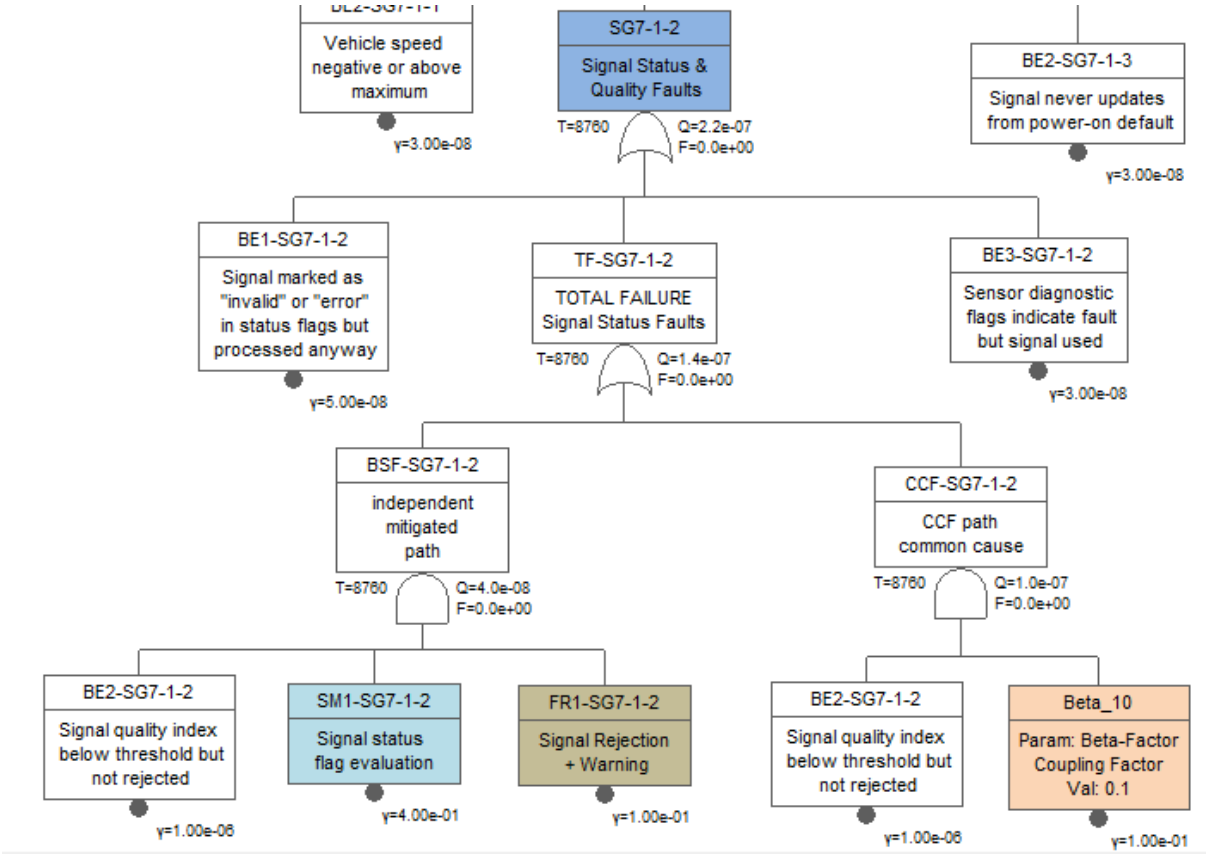
- **Linker Ast (Unabhängig):** Die oben beschriebene Kette aus Fehler, Detektion und Reaktion.
- **Rechter Ast (Common Cause):** Ein Pfad, der die Sicherheitsmechanismen umgeht.

Dynamische Berechnung des CCF-Pfades:
Um Konsistenz zu gewährleisten, wird der CCF-Pfad nicht mit einem statischen Wert befüllt. Stattdessen nutzen wir ein UND-Gatter, das zwei Eingänge verknüpft:

1. **Das originale Base Event:** (Referenz auf das Ereignis im linken Ast).
2. **Der Beta-Faktor:** Ein Parameter-Event, das den Grad der Kopplung beschreibt (z.B. $\beta = 0{,}1$ für Shared Resources ohne MPU).

Dies stellt sicher: Wenn die Software-Qualität des Moduls verbessert wird (Occurrence sinkt), sinkt automatisch auch das Risiko im Common-Cause-Pfad.

Beispiel:



4.3 Modellierung externer Einflüsse

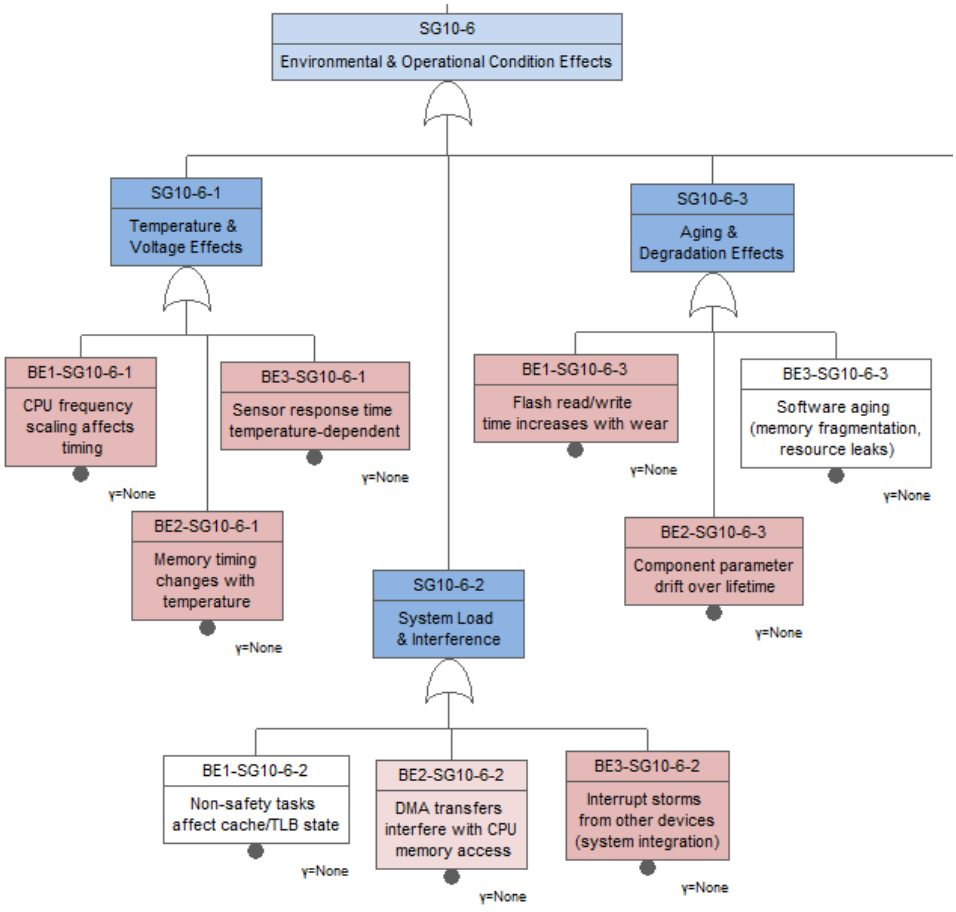
Neben der internen Plattform-Kopplung existieren externe Stressoren, die Software-Verhalten indirekt beeinflussen (z.B. CPU-Drosselung durch Temperatur, EMV-Störungen auf Bus-Leitungen). Da diese Ursachen bekannt sind, werden sie nicht pauschal über einen Beta-Faktor erschlagen, sondern explizit modelliert (**Explicit Modeling**).

Der Umwelt-Baum:
Wir erstellen separate Fehlerbäume für Phänomene wie:

- **Physical Stress:** Temperatur, Unterspannung.
- **Runtime Degradation:** Speicherfragmentierung, Resource Leaks.
- **System Interference:** Interrupt-Storms, DMA-Konflikte.

Diese Bäume werden mittels **Transfer-Gattern** in die betroffenen funktionalen Fehlerbäume eingespeist. Sie wirken in der Regel als ODER-Verknüpfung auf das Top-Level: Egal wie gut die Software-Logik ist, wenn die Hardware-Basis durch externe Einflüsse instabil wird, fällt die Funktion aus.

Beispiel:



4.4 Plausibilisierung

Der aktualisierte Fehlerbaum liefert nun den **RPS Final** für das Gesamtsystem unter Berücksichtigung aller Abhängigkeiten.

Hier zeigt sich oft der dominante Einfluss der DFA:

- Ein Pfad mag in der SDRM (Excel) „grün“ erscheinen (z.B. Score 40).
- Durch den Beta-Faktor (z.B. 10% Kopplung) entsteht jedoch ein paralleler Risikopfad mit einem Score von 100.
- Das Gesamtergebnis (140) zeigt, dass die Architektur trotz Mechanismen unsicher ist.

Diese Plausibilisierung zwingt den Architekten dazu, nicht nur bessere Algorithmen zu schreiben, sondern die **Architektur zu härten** (z.B. Einsatz von Memory Protection Units, Separierung auf unterschiedliche Cores oder diversitäre Redundanz), um den Beta-Faktor zu senken.

5. Phase 5: Verhaltensanalyse mittels ETA (Event Tree Analysis)

Während die FTA (Phase 1-4) die logischen Ursachen eines Fehlers analysiert („Warum tritt das Ereignis ein?“), untersucht die Event Tree Analysis (ETA) die Konsequenzen dieses Ereignisses („Was passiert danach?“).

In der Software-Sicherheitsanalyse nutzen wir die ETA nicht, um Fahrzeugunfälle zu simulieren, sondern um die **Wirksamkeit der Sicherheitsarchitektur** zu validieren. Wir prüfen, ob die Kette aus Architektur-Integrität, Detektion und Reaktion robust genug ist, um einen auftretenden Software-Fehler in den **Safe State** zu leiten.

5.1 Sequenzielle Betrachtung

Ein Fehlerbaum kennt keine Zeit. Ein UND-Gatter sagt nur, dass zwei Bedingungen gleichzeitig wahr sein müssen. In der Realität der Software-Ausführung ist die **zeitliche Abfolge** jedoch entscheidend.

Die ETA modelliert diese Sequenz. Ein Pfad im Ereignisbaum stellt den zeitlichen Ablauf dar:

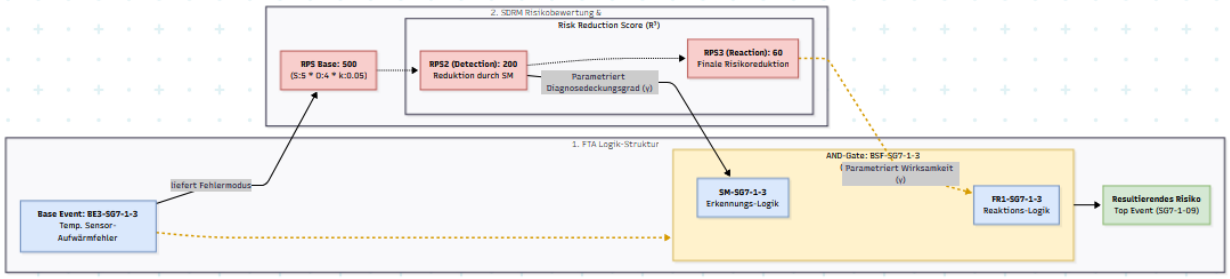
1. **Initiating Event:** Der Software-Fehler tritt auf (Startpunkt).
2. **Barriere 1:** Hält die Architektur stand (kein Common Cause)?
3. **Barriere 2:** Schlägt die Detektion an?
4. **Barriere 3:** Wird die Reaktion erfolgreich ausgeführt?

Diese Betrachtung zwingt den Architekten zu prüfen: *„Ist sichergestellt, dass die Reaktion erst erfolgt, wenn die Detektion abgeschlossen ist?“* und *„Kann die Reaktion ausgeführt werden, bevor der Fehler zur Gefährdung führt (Fault Tolerant Time Interval)?“*

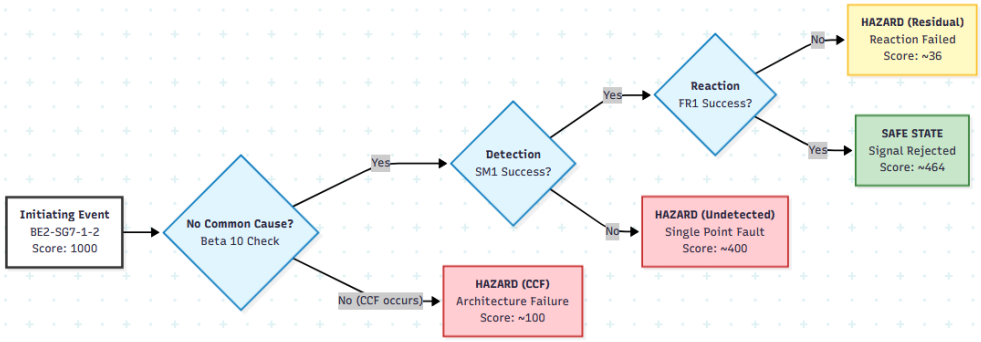
5.2 Das Bow-Tie-Modell: Verknüpfung von Ursachenanalyse (FTA) und Konsequenzanalyse (ETA).

Methodisch verknüpfen wir die Phasen 1-4 (FTA) und Phase 5 (ETA) im sogenannten **Bow-Tie-Modell** (Krawatten-Modell).

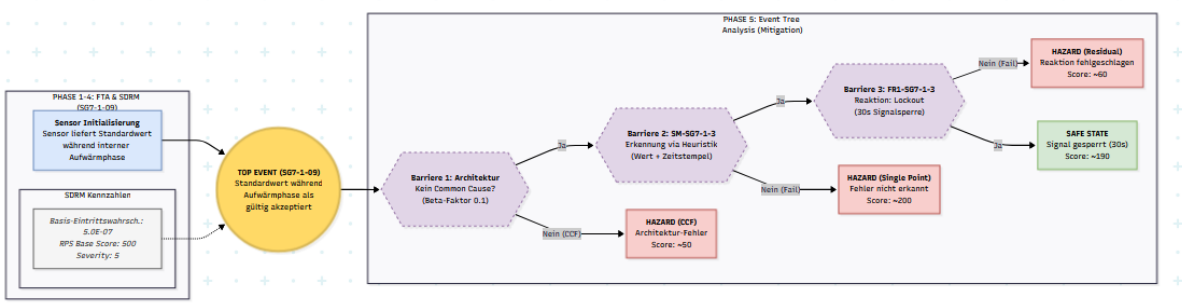
- **Der Knoten (Mitte):** Das **Top Event** aus der FTA. Dies ist gleichzeitig das **Initiating Event** der ETA.
- **Die linke Seite (Ursachen):** Der Fehlerbaum mit allen Base Events, Beta-Faktoren und Umwelt-Einflüssen. Hier berechnet sich der **Input-Score** (die Häufigkeit/Wahrscheinlichkeit, mit der das System in den kritischen Zustand gerät).



- **Die rechte Seite (Folgen):** Der Ereignisbaum. Er verteilt den Input-Score auf die verschiedenen Endzustände (Sicher vs. Unsicher).



Das Bow-Tie-Modell dient als Gesamtübersicht der Sicherheitsargumentation: Links minimieren wir die Ursachen, rechts maximieren wir die Beherrschbarkeit.



5.3 Bewertung der Pfade

Die ETA dröselt das Gesamtrisiko in spezifische Szenarien auf. Startpunkt ist das Initiating Event mit seinem **RPS Base** (z.B. 1000 Punkte). Der Baum verzweigt sich an den Barrieren, wobei jede Abzweigung Wahrscheinlichkeitsmasse (Score) in eine bestimmte Richtung lenkt.

Wir unterscheiden vier typische Endzustände (End States) in der Software-ETA:

1. Hazard (CCF / Architectural Break-Through)

- *Pfad:* Der Fehler tritt auf **UND** die Architektur-Unabhängigkeit versagt (Common Cause).
- *Bedeutung:* Der Fehler im Code reißt den Sicherheitsmechanismus sofort mit (z.B. durch Speicherüberschreibung). Die Detektion kommt gar nicht zum Einsatz.
- *Rechnung:* $RPS_{Input} \times \beta$

2. Hazard (Undetected / Single Point Fault)

- *Pfad:* Kein CCF **UND** Detektion versagt.
- *Bedeutung:* Die Architektur hält, aber der Diagnose-Algorithmus (z.B. Range Check) erkennt den spezifischen Datenfehler nicht (Lücke in der DC).
- *Rechnung:* $RPS_{Input} \times (1 - \beta) \times (1 - DC)$

3. Hazard (Residual Fault)

- *Pfad:* Kein CCF **UND** Detektion erfolgreich **UND** Reaktion versagt.
- *Bedeutung:* Der Fehler wurde erkannt, das System versucht in den Safe State zu schalten, aber die Reaktions-Logik (z.B. Umschalten auf Ersatzwert) schlägt fehl.
- *Rechnung:* $RPS_{Input} \times (1 - \beta) \times DC \times (1 - R^3)$

4. Safe State (Erfolgs-Pfad)

- *Pfad:* Kein CCF **UND** Detektion erfolgreich **UND** Reaktion erfolgreich.
- *Bedeutung:* Das System hat einen Fehler, fängt ihn aber kontrolliert ab (z.B. Deaktivierung der Funktion, Nutzung von Default-Werten). Es tritt keine Gefährdung auf.
- *Rechnung:* Der Rest des Scores landet hier.

Das Ziel der Analyse:
Die ETA liefert den quantitativen Nachweis der **Wirksamkeit**. Sie zeigt, dass der Großteil des Risikos (z.B. 99,9% des Scores) im Ast „Safe State“ landet. Die verbleibenden Rest-

Scores in den Hazard-Ästen müssen in der Summe unter dem akzeptierten Schwellenwert liegen.

Damit ist die Sicherheitsanalyse abgeschlossen: Wir haben hergeleitet, woher der Fehler kommt, wie wir ihn entdecken und bewiesen, dass er sicher abgefangen wird.

6. Kritische Auseinandersetzung

Jede Methodik ist ein Werkzeug, und kein Werkzeug ist für jeden Einsatzzweck perfekt. Der hier vorgestellte hybride Ansatz aus FTA, SDRM und ETA stellt einen signifikanten Fortschritt gegenüber der isolierten SW-FMEA dar, bringt aber eigene Herausforderungen mit sich. Dieses Kapitel beleuchtet Stärken, Risiken und die Abgrenzung zu etablierten Alternativen.

6.1 Methodische Stärken

Der Hauptnutzen dieses Vorgehens liegt in der **Effizienz durch Fokussierung**.

1. **Reduktion der Komplexität:**
   Durch die Einführung des RPS-Schwellenwertes in Phase 2/3 (SDRM) wird das „Rauschen" herausgefiltert. Anstatt Ressourcen darauf zu verwenden, tausende triviale Fehlermöglichkeiten zu dokumentieren, konzentriert sich die Entwicklungsarbeit auf die kritischen Pfade (die „Top-Scorer"). Dies entspricht dem Pareto-Prinzip in der Sicherheitsanalyse.

2. **Architektur-Validierung statt Dokumentation:**
   Die klassische FMEA ist oft eine reine Fleißarbeit, die *nach* der Entwicklung erfolgt. Die hier genutzte Rückführung in die FTA (Phase 4) und die DFA-Betrachtung zwingen den Entwickler jedoch dazu, die **Architektur zu härten**. Das Sichtbarmachen von Common Cause Failures (Beta-Faktor) deckt Schwachstellen in der Plattform auf, die in einer tabellarischen Liste unsichtbar bleiben würden.

3. **Nachweis der Wirkungskette:**
   Die Kombination aus Detektion und Reaktion („No Detection without Intention") und deren Validierung in der ETA (Phase 5) schließt die argumentative Lücke. Es wird nicht nur behauptet, dass ein Mechanismus existiert, sondern nachgewiesen, dass er im zeitlichen Ablauf effektiv zum sicheren Zustand führt.

6.2 Grenzen und Risiken: Umgang mit semi-quantitativen Scores, Konsistenzsicherung zwischen Modellen.

Die Anwendung dieser Methode erfordert Disziplin und ein Bewusstsein für methodische Fallstricke.

1. **Das Risiko der Scheingenauigkeit (False Precision):**
   Die Verwendung von RPS-Werten (z.B. „1000", „40") und Formeln verleitet dazu, diese als mathematische Wahrheit zu betrachten.

   - *Kritik:* Software hat keine Physik. Die Werte sind **Experten-Schätzungen** (Expert Judgement), keine statistischen Messwerte.

- *Mitigation:* Es muss allen Beteiligten klar sein, dass der RPS ein **relatives Priorisierungsmaß** ist, keine absolute Ausfallwahrscheinlichkeit. Ein Score von 40 ist nicht „zweimal so sicher“ wie 80, sondern **er liegt schlicht im akzeptierten Bereich.**

2. **Datenkonsistenz („Tooling Gap“):**
   Der Prozess erfordert den Datenaustausch zwischen verschiedenen Sichten: Baumstruktur (FTA/ETA) und Tabelle (SDRM).
   - *Risiko:* Ohne integrierte Tool-Landschaft (oder Skripte) drohen Inkonsistenzen. Wenn sich ein Base Event in der Architektur ändert, muss es manuell in der SDRM und im Beta-Faktor-Ast der FTA nachgezogen werden, was jedoch auch für andere Methoden gültig ist.
   - *Mitigation:* Strikte Anwendung des „Single Source of Truth“-Prinzips (z.B. Referenzierung von Base Events im FTA-Tool) und Disziplin im Change Management.
3. **Komplexität der DFA:**
   Die Modellierung des Beta-Faktors ist abstrakt. Die Festlegung, ob β nun 5% oder 10% beträgt, ist oft diskussionswürdig und schwer zu belegen. Hier ist eine konservative Auslegung zwingend erforderlich, um das Risiko nicht schönzurechnen.

6.3 Abgrenzung zu anderen gängigen Analysemethoden

Wie positioniert sich dieser Ansatz im Vergleich zu den etablierten Industriestandards? Die folgende Tabelle stellt die Schwächen der herkömmlichen Methoden unserer Lösung gegenüber.

Etablierte Methode	Typischer Fokus	Schwäche der etablierten Methode	Lösung im hybriden Ansatz
Klassische SW-FMEA	Induktiv (Bottom-Up). Zeilenweise Analyse von Funktionen/ Modulen.	**Ineffizienz:** Verliert oft den Bezug zum Sicherheitsziel. Sehr hoher Dokumentationsaufwand für unkritische Pfade.	**Fokussierung:** Wir arbeiten deduktiv (Top-Down via FTA). Durch den RPS-Filter werden irrelevante Pfade *vor* der Detailarbeit aussortiert.
STPA (System-Theoretic Process Analysis)	Fokus auf Interaktionen, Regelkreise und unsichere Kontrollaktionen.	**Abstraktions-Lücke:** Exzellent auf Systemebene, aber oft schwer auf konkrete Software-Variablen und Code-Zeilen herunterzubrechen.	**Konkretisierung:** Unser Ansatz ist „Code-näher“ und für Software-Architekten greifbarer, da er direkt auf Komponenten, Variablen und Interfaces aufsetzt.
Reine FTA (Qualitativ)	Logische Analyse von Fehlermodi in Baumstruktur.	**Unübersichtlichkeit:** Wird bei Software schnell unlesbar, wenn man versucht, Attribute (DC-Werte, Reaktionstypen) direkt im Baum zu verwalten.	**Daten-Management:** Wir nutzen die FTA nur für die Logik. Die Verwaltung der Attribute und Bewertungen lagern wir in die SDRM (Tabelle) aus.

Fazit:
Der hybride Ansatz positioniert sich als **„Pragmatische Mitte“**. Er nutzt die logische Strenge der FTA für die Struktur und die tabellarische Übersicht der FMEA (in Form der SDRM) für die Bewertung, ergänzt um die notwendige Tiefe der DFA/ETA, die den modernen Sicherheitsstandards gerecht wird.

7. Zusammenfassung und Ausblick

Die Entwicklung sicherer Software ist kein bürokratischer Akt, sondern eine Disziplin der Architektur. Das in diesem eBook vorgestellte Vorgehen bricht bewusst mit der Tradition der rein dokumentarischen SW-FMEA, die oft erst entsteht, wenn der Code bereits geschrieben ist.

Wir haben einen hybriden Ansatz vorgestellt, der die logische Tiefe der **Fehlerbaumanalyse (FTA)** mit der systematischen Bewertung einer **SDRM (Safety Design & Rating Matrix)** und der Verhaltensvalidierung der **Event Tree Analysis (ETA)** verbindet.

7.1 Die Kern-Ergebnisse

Durch die Anwendung dieses 5-Phasen-Modells erreichen wir drei wesentliche Ziele, die mit herkömmlichen Methoden nur schwer realisierbar sind:

1. ### Fokussierung statt Masse
   Durch den deduktiven Einstieg (Top-Down FTA) und die Einführung des RPS-Filters in der SDRM (Phase 2) reduzieren wir den Analyseaufwand drastisch. Wir analysieren nicht *alles*, sondern nur das, was die Sicherheitsanforderung logisch gefährdet. Das Verhältnis von initialen Events zu tatsächlich zu behandelnden Pfaden schont Entwicklungsressourcen.

2. ### Architektur als Zentrum
   Sicherheit entsteht nicht durch Code-Zeilen, sondern durch Architektur-Entscheidungen. Die explizite Modellierung von **Common Cause Failures** (via Beta-Faktor) und **Umwelteinflüssen** (via Transfer-Bäume) in Phase 4 macht die Abhängigkeiten der Plattform sichtbar. Sie zwingt den Architekten, über Entkopplung (**Freedom from Interference**) nachzudenken, statt nur auf algorithmische Korrektheit zu vertrauen.

3. ### Durchgängige Argumentationskette
   Das Bow-Tie-Modell (Phase 5) schließt die Lücke zwischen Ursache und Wirkung. Wir beweisen nicht nur, dass wir einen Fehler erkennen (Detektion), sondern weisen mittels ETA nach, dass die darauf folgende Reaktion das System effektiv in den sicheren Zustand überführt.

7.2 Der Prozess auf einen Blick

Zusammenfassend lässt sich der Workflow als geschlossener Regelkreis („Closed Loop“) beschreiben:

- **Phase 1 (Logik):** Ableitung der qualitativen FTA direkt aus der SW-Architektur und den negierten Sicherheitsanforderungen.

- **Phase 2 (Bewertung):** Übertrag in die SDRM und Berechnung des Basis-Risikos (RPS).
- **Phase 3 (Design):** Filterung kritischer Pfade und Definition von Detektion und Reaktion („No Detection without Intention“).
- **Phase 4 (Validierung):** Rückführung in die FTA, Ergänzung der Sicherheitsmechanismen und Härtung gegen interne Kopplungen (Beta-Faktor).
- **Phase 5 (Nachweis):** Sequenzielle Analyse der Wirksamkeit mittels ETA.

7.3 Ausblick: Der Weg zur Automatisierung

Die hier beschriebene Methodik ist prädestiniert für einen höheren Automatisierungsgrad. Da die FTA strikt der Architektur folgt, lassen sich initiale Bäume – wie in Phase 1 beschrieben – zunehmend durch **KI-Modelle (LLMs)** generieren.

Der nächste logische Schritt in der Evolution der Sicherheitsanalyse ist die technische Integration der Werkzeuge. Der manuelle Übertrag zwischen FTA-Tool (z.B. Arbre Analyst) und Tabellenkalkulation (SDRM in MSExcel) ist fehleranfällig. Zukünftige Tool-Ketten werden diesen „Roundtrip“ mittels Scripting automatisieren, sodass Änderungen in der Architektur (z.B. ein neuer Schnittstellen-Parameter) automatisch einen Ast im Fehlerbaum erzeugen und eine Zeile in der Bewertungsmatrix flaggen.

Schlusswort
Sicherheit ist kein Add-on. Mit dem hier vorgestellten hybriden Ansatz wird Software Safety Analysis zu einem integralen Bestandteil des Software-Designs – effizient, logisch fundiert und architekturgetrieben.