Kolloquium

Ein Konzept für die Testfallentwicklung für sicherheitskritische Anforderungen unter Verwendung von Fault-Injection und Mutationstests

Ein Vortrag von Benjamin Trapp
(30.04.2014)

Agenda

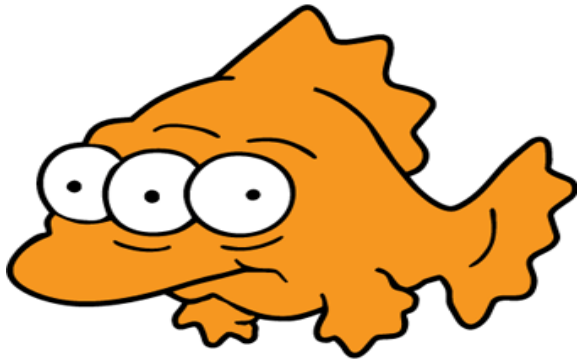
„Es irrt der Mensch, solang er strebt.“ (Johann W. von Goethe)

1. Grundlagen
2. Architektur der Implementierung
3. Testszenarien
4. FI-Experiment: Szenario Kosmische Strahlung
5. Fazit und Kritik

Mutationstest – Was ist das?

- Fehlerbasierte Testmethode (White-Box-Test)
- Bewertet die Qualität der Testfälle anhand von Mutanten
- **Mutant**: Softwareversion (mit jeweils nur einem Fehler), die unter Verwendung einer Mutationstransformation aus der Originalversion abgeleitet wird
- **Mutationstransformation**: Vordefinierte Regel die zu einem bestimmten Programmfehler führt. Die Regel basiert üblicherweise auf „Vertauschen“ von Einzelanweisungen z.B. *if* (*a* >= *b*){…} ➔ *if* (*a* > *b*){…}

Ziel: Aufspüren von „schwachen Tests“



Mutationstests - Ablauf

Original Quellcode

```
if ( a && b)
      c = 1;
else
      c = 0;
```

Mutierter Quellcode

```
if ( a || b)
      c = 1;
else
      c = 0;
```

Durchführung einer einzelnen Mutation

Ausführung der Tests

Test “tötet“ den Mutant
➔Test adäquat

Mutant überlebt
➔Test inadäquat und muss verstärkt werden

Mutationstests– Schwache Tests


```
public void String foobar(int i) {
  if ( i >= 0 )
     return "foo“;
 else
     return "bar“;
}
```

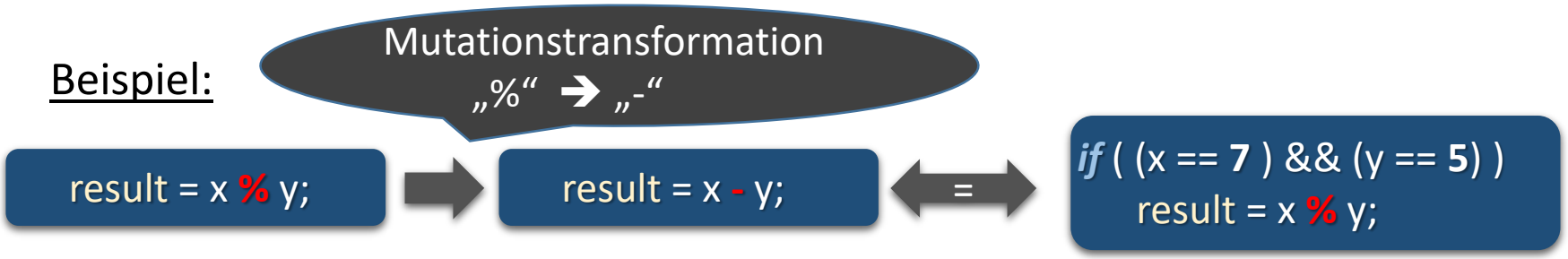
```
@Test
public void shouldReturnFooWhenGivenMinus1(){
     assertEquals("foo“ , foobar(-1));
}

@Test
public void shouldReturnBarWhenGiven1(){
     assertEquals("bar“ , foobar(1));
}
```

Gründe für inadäquate Tests

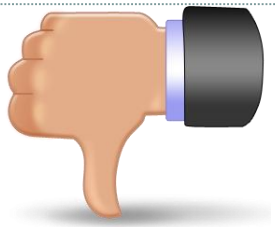
Mutant ist inadäquat und bleibt unerkannt weil:

- Testfälle ungeeignet um den Fehler zu identifizieren
  ➔ Verbesserung oder ggf. Erstellung neuer Testfälle notwendig
- Es existiert kein Testfall der in der Lage ist den Mutanten zu erkennen
  ➔ Mutant ist äquivalent zum Originalprogramm
- **Äquivalenz**: Mutant unterscheidet sich syntaktisch vom Originalprogramm weißt jedoch die gleiche funktionale Semantik auf

Beispiel:



Mutationstests – Stärken & Schwächen



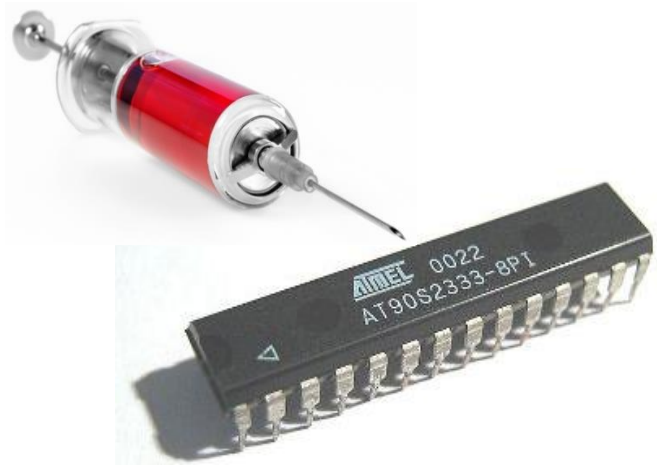
- Mutationstests sind teuer ➔ Sehr hoher Aufwand beim Kompilieren, Ausführung und Vergleich
- Zeitaufwendige Reduzierung der überlebenden Mutanten

- Bieten Gewissheit über die Qualität der Testdaten ➔ verbessern indirekt die Softwarequalität
- Besonders stark in Kombination mit test-driven development (TDD)
- Gut automatisierbar ➔ Problemlose Integration in Softwarelieferprozesse wie Contious Delivery

Fault-Injection – Was ist das?

- Relevante Methode zum Testen von funktionalen Sicherheitsstandards
- Durchführung der Fehlereinpflanzung mittels hinzufügen diverser Fehler in unterschiedlichen Bereichen des Softwaresystems
- Fehlerbehandlung von besonderer Bedeutung ➔ Stellt die Robustheit des Softwaresystems und der Fehlerbehandlungsstrategie sicher
- Fault-Injection dient neben Robustheitstest auch als Stresstest
- Drei Varianten: Hardware-, Software- und Simulierte Fault-Injection

Ziel: Überprüfung von schwer test- oder reproduzierbare Schwachstellen



Software Fault-Injection

- Testtechnik zur Erforschung des Verhaltens von Software, wenn diese mit unüblichen Methoden strapaziert wird ➔ Einpflanzen von Fehler in den Code
- Hilft die Testabdeckung zu verbessern
- Insbesondere Codestellen zur Fehlerbehandlung die üblicherweise nicht ausgelöst werden hier überprüft
- Verfügt typischerweise über zwei Methoden zur Fehlereinzupflanzung:

Compile-Time Injections
Technik bei der, der Fehler bereits vor dem Kompilieren eingepflanzt wird. Fehler ist statisch und kann sich zur Laufzeit nicht mehr verändern.

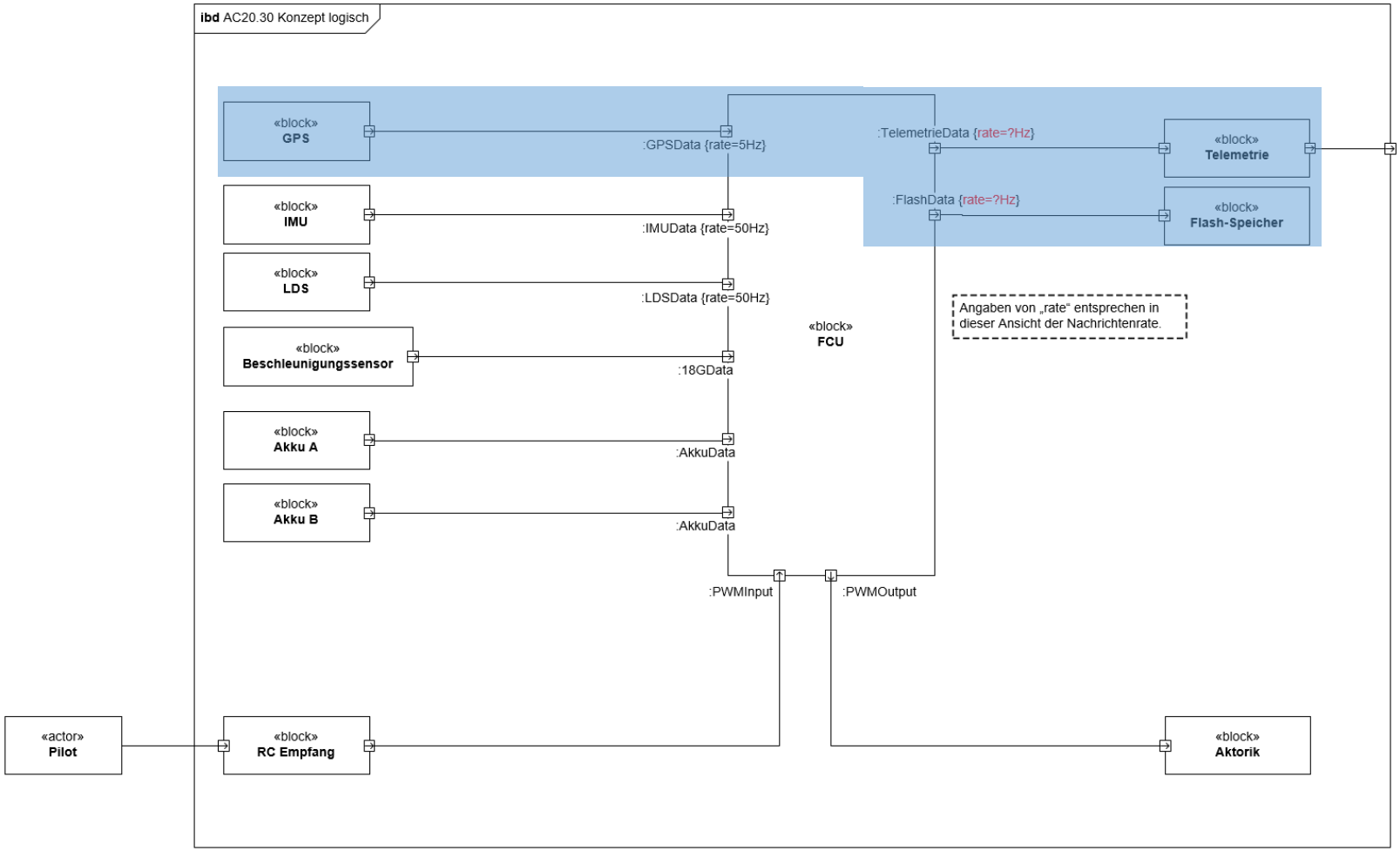
Run-Time Injections
Benützt Software-Trigger um Fehler während der Laufzeit in ein Softwaresystem einzupflanzen. Die Trigger sind typischerweise Zeit- oder Interruptgesteuert.

Virtual Hardware in the Loop (vHIL)

Ziel: Frontverlagerung des Testprozesses durch den Einsatz eines virtuellen Prototyps

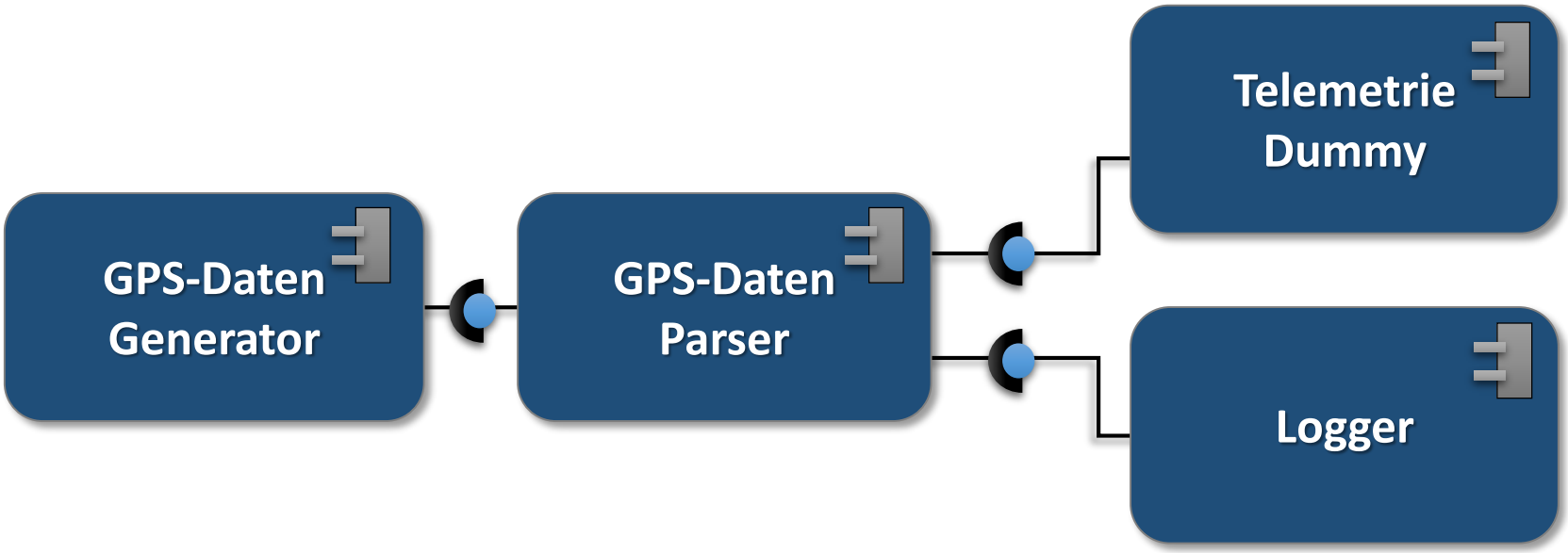
- **Virtueller Prototyp**: Softwaremodell das die zu erstellende reale Hardware emuliert ➔ Bietet die Option, die identische binäre Software auszuführen die später auf der realen Hardware läuft (ohne das Modell zu verändern)
- Für die Emulation reicht (in der Regel) ein herkömmlicher Desktop-PC
- Versetzt Softwareteams in die Lage, anhand eines realistischen Abbilds der Hardware bereits vor Fertigstellung der realen Hardware zu beginnen
- Durch diese Frontverlagerung können Fehler frühzeitig gefunden werden ➔ Steigert signifikant die Qualität des Endprodukts bei gleichzeitiger Reduzierung der Kosten

Architektur des Flight Control Unit



Quelle: René Büscher (Konzept_IBD.pdf)

Architektur des virtuellen Prototyps



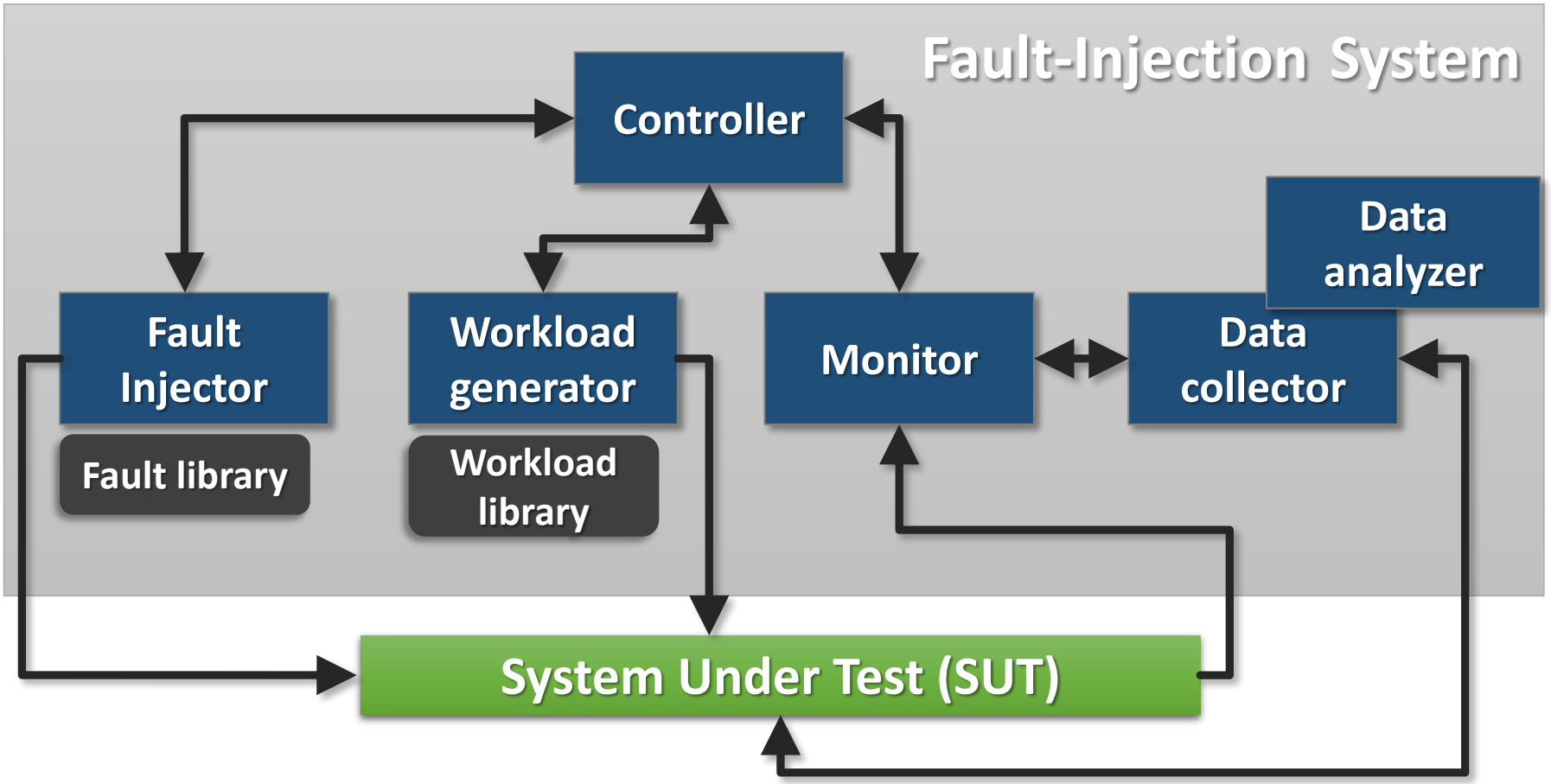
NMEA-Datensätze - Aufbau

- **<$GP><- - ->,<X>,...,<Xn>*<Checksum><CR><LF>**
- **<$GP>** Talker ID ➔ ($GP für GPS, $GL für GLONAS)
- **<- - ->** Name des Datensatzes z.B. RMC, GGA, DTM, …
- **<x>,…,<Xn>** Vordefinierte Menge an Daten die im Kontext des Datensatzes stehen
- ***<Checksum>** Berechnung der Checksumme
- **<CR><LF>** Zeilenumbruch als Terminierungszeichen (**CR** = Carriage Return, **LF** = Linefeed)

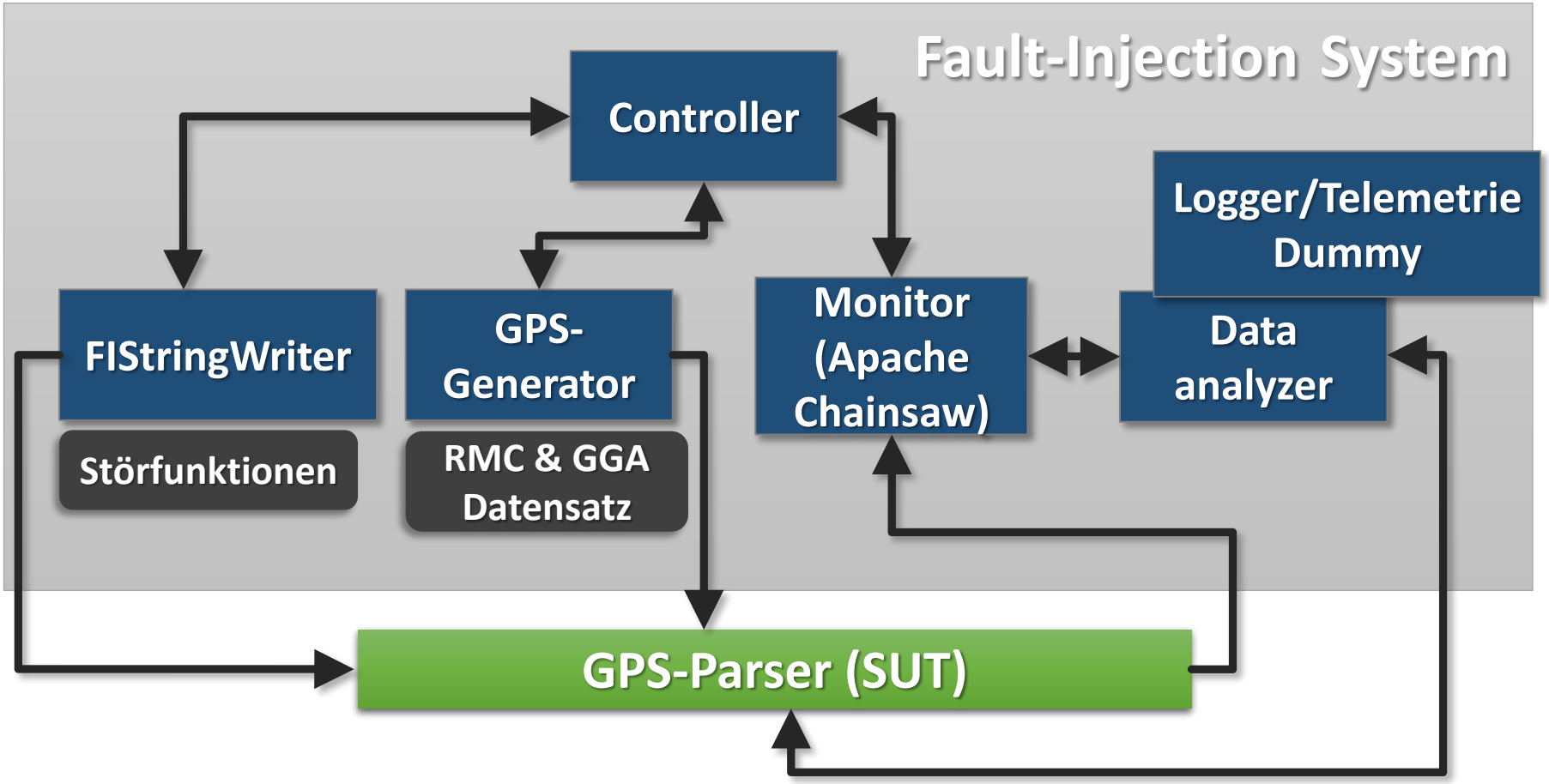
Beispiel:

```
$GPRMC,195923,A,5335.62,S,1004.12,W,010.0,203.0,120314,,S*73
$GPGGA,195922,5335.62,S,1004.12,W,8.0,08,2.2,13.1,M,0,M,,*79
$GPVTG,0.7,T,0.8,M,13.0,N,24.0,K,D*2D
$GPDTM,P92,,0.0,,0.0,,0.0,P92*64
```

Fault-Injection Environment



Angepasste Fault-Injection Environment



Testszenarien (1/4)

- Erprobung von Konzepten innerhalb des Fault-Injection Experiments
- Formen die Randbedingungen des FI-Experiments
- Dient nicht zum Auffinden von Fehlern, sondern als Grundlage zum Ableiten der einzupflanzenden Fehlertypen
- Abstrahierte Nachempfindung der implementierten Fehlertypen von realen Fehlerquellen ➔ Stellen Bezug zur Realität her
- Testszenarien als Prüfkonzept für Fehlerbehandlungsmechanismen ➔ Nachweis über Robustheit und Funktionalität

Ziel: Überprüfung und Immunisierung der Soft- und Hardware gegenüber den eingepflanzten Fehlertypen

GPS-Spoofing (2/4)

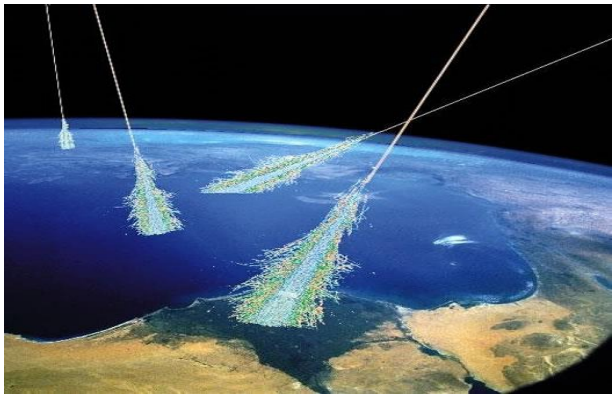
- Angreifer sendet Störsignale (typischerweise DGPS) welche die regulär empfangenen GPS-Signale des Ziels imitieren
- Keine Störung im eigentlichen Sinn ➔ Täuschung des GPS-Empfängers durch falsche Positionsangaben
- Auftreten von „Sprüngen“ in den Daten für Längen-, Höhen und oder Breitengrad in den empfangenen NMEA-Datensätzen
- „Fehler“ gehört eigentlich zum Bereich Safety (Blick über den Tellerrand)

Testszenarien – Haftfehler (3/4)

- Haftfehler ➔ Annahme, dass ein Ein- bzw. Ausgang oder eine Leitung auf einem festen Wert liegt und keinerlei Signalwechsel möglich sind
- Wird in Hardware typischerweise durch Materialfehler ausgelöst
- Haftfehler verursacht durch kosmische Strahlung ebenfalls denkbar ➔ Datum innerhalb der Software wird während des Zustandswechsels verfälscht und erreicht ungültigen Zustand ➔ Extern als Haftfehler beobachtbar
- Hat besondere Bedeutung für Hardwarebeschreibungssprachen, wie bspw. VHDL, Verilog
- Ebenfalls stark in Verbindung mit Modellierungs- und Simulationssprachen/Tools, z.B. SystemC, Simulink

Testszenarien – Kosmische Strahlung (4/4)

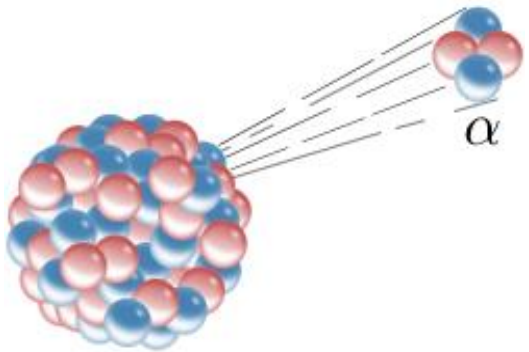
- Untersucht die Auswirkungen von radioaktiver- und kosmischer Strahlung auf das Fehlerverhalten von Hard- und Software
- **Soft Error**: Üblicherweise Strahlung als Auslöser. Aber auch Störsignale wie Rauschen oder Übersprechen von Signalen gehören in diese Kategorie
- Hardware ➔ Strahlenbasierte Fault-Injection bereits üblicher Standard zur Qualitätssicherung



Kosmische Teilchenschauer auf dem Weg zur Erde

Kosmische Strahlung

- Hochenergetische Teilchenstrahlung aus dem Weltall
- In Bezug auf Soft-Errors ➔ α-Strahlung von besonderer Bedeutung
- α-Teilchen sind zweifach positiv geladene Helium Ionen
- Die hohe elektrische Ladung des energiereichen Helium Ions, erzeugt freie Ladungsträger im Mikrochip ➔ Führt zu kleinen elektrischen Stromflüsse
- Entstehung von „Bitflips“ ➔ Funktionsfehler oder gar Totalausfall des Chips



Emission eines α-Teilchens

Schutz gegen kosmische Strahlung

- Aufteilung in zwei Bereiche zum auffinden und ggf. Korrektur von Fehlern:
  1. Verwendung von Checksummen.
  2. Softwaremethoden zur Fehlerkorrektur im Datenbereich.
- Literatur über „Softwaremethoden zur Fehlerkorrektur im Datenbereich“ Relativ dünn gesät.

  Beispiel: Swift-R: Drei Programmkopien ineinander kombiniert. Mehrheitswahl vor kritischen Entscheidung
- Im Bezug auf des Testszenarios
  ➔ Verwendung der NMEA-Checksumme
- Einsatz der FI zur Bewertung von Effektivität und Robustheit

NMEA-Checksumme

- NMEA-Checksumme wird durch Verwendung von XOR gebildet
- Triviales Checksummenverfahren
- Leider oft Deaktiviert ➔ Verwendung optional und kein muss

```
public String NMEAChecksum ( String  nmeaSentence){
      … Fehlerbehandlung …

    //i == 1 um das '$' Zeichen zu überspringen
   for (int i = 1; i < nmeaSentence.length(); i++)
                        chk ^= nmeaSentence.charAt(i);

    … String Bearbeitung und Return Statement …
}
```

Störfunktionen

- Erzeugt den Fehler für die Fehlereinpflanzung ➔ Methode die den gewünschten Fehlertyp generiert
- Verändern die Daten auf die das Programm Zugriff hat
- Verändert den Programmcode selbst nicht
- Die Möglichkeiten für die Erstellung von Störfunktionen ist nahezu unbegrenzt

Beispiele:

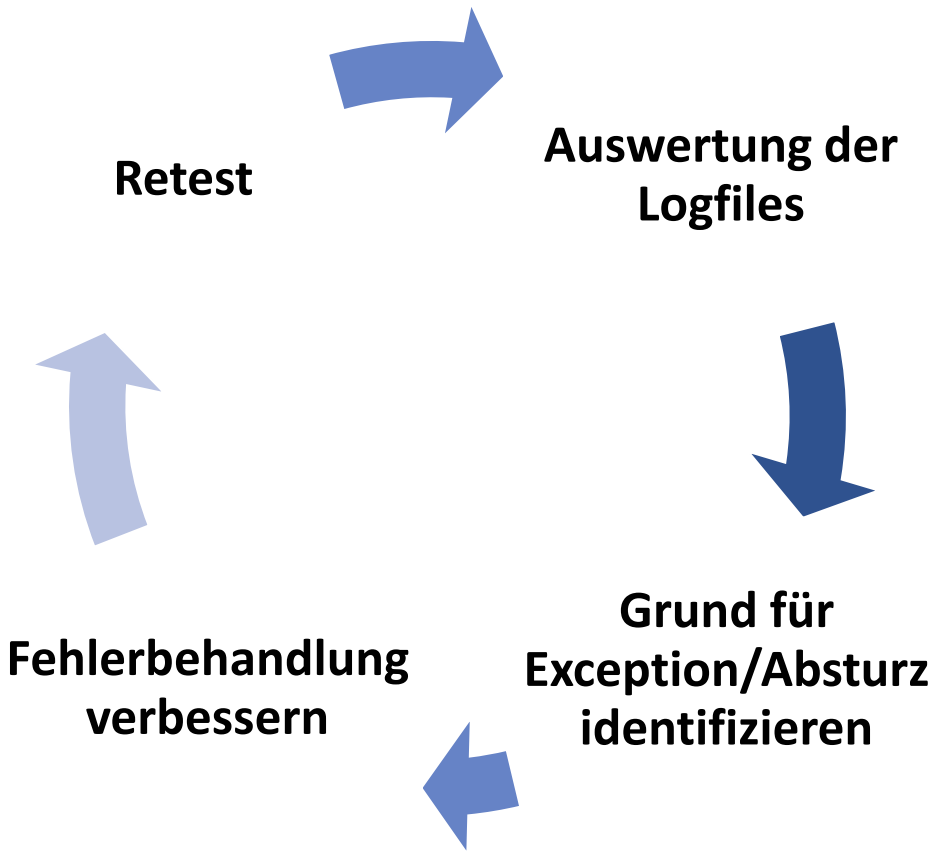
```
public int offByOne(int  var) {
    return (var % 2 == 0)? ++var : --var;
}
public int flipBits(int  var, int bits) {
    return var ^(1 << bits);
}
```

Durchführung des FI-Experiments

- Simulation einer Störung bei der Datenübertragung (RS422)
- **Verwendete Störfunktion**: Zufälliger Austausch einzelner übertragener „Symbole“ durch randomisierte ASCII-Symbole

MARKER	LEVEL	LOGGER	MESSAGE
	INFO	Unknown	$GPGGA,195909,5334.04,S,1001.97,W,8.0,02,2.1,14.8,M,0,M,,*7E
	ERROR	Unknown	NMEA-Sentence malformed ($&PGGA,195910,5334.16,S,100&.19,E,8.0,04,2.0,14.7,M,¥,M,,*6A)
Malformed	INFO	Unknown	$&PGGA,195910,5334.16,S,100&.19,E,8.0,04,2.0,14.7,M,¥,M,,*6A
	ERROR	Unknown	NMEA-Sentence malformed (^GPR(C,195910,A,5334.16,S,1 02.19,E,010.0,140.0,120314é,SV6A)
Malformed	INFO	Unknown	^GPR(C,195910,A,5334.16,S,1 02.19,E,010.0,140.0,120314é,SV6A
	INFO	Unknown	$GPGGA,195911,5334.23,S,1002.4,E,8.0,06,2.0,14.5,M,0,M,,*51
	ERROR	Unknown	NMEA-Sentence malformed ($öPGGA,195912,5334.3,S,100D.48,E,8.0,10,2.ã,14.2,M,0,M,,*ô8)
Malformed	INFO	Unknown	$öPGGA,195912,5334.3,S,100D.48,E,8.0,10,2.ã,14.2,M,0,M,,*ô8
	INFO	Unknown	$GPRMC,195912,A,5334.3,S,1002.48,E,010.0,103.0,120314,,S*5F
	INFO	Unknown	$GPGGA,195913,5334.46,S,1002.71,W,8.0,03,2.1,13.8,M,0,M,,*7E

Auswertungsprozess des FI-Experiments



Fazit und Kritik

Der Zuwachs an Weisheit lässt sich genau nach der Abnahme an Galle bemessen (Friedrich Nietzsche)

- Kein Allheilmittel ➔ Beide vorgestellten Testmethoden reichen als „alleinige“ Testverfahren nicht aus
- Fehlermodelle nur so gut wie ihr Konzept ➔ mit latenten Softwarefehler muss dennoch gerechnet werden (Mensch übersieht gerne Probleme)
- FI nur stark durch Verwendung unterschiedlicher FI-Varianten und Fehlermodelle ➔ Variierte Abstraktionsniveaus der Experimente
- Für kleinere Modelle ➔ Simulation mit Matlab Simulink besser geeignet
- Generische Gestaltung und Automatisierung teilweise schwer, jedoch sind Konzepte wie der „Chaos Monkey“ von Amazon Web-Services vielversprechend

Vielen Dank für Eure Aufmerksamkeit!