Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Benjamin-Yves Johannes Trapp

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

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Benjamin-Yves Johannes Trapp

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

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Bettina Buth
Zweitgutachter: Prof. Dr.rer.nat. Thomas Lehmann

Eingereicht am: 14. April 2014

Benjamin-Yves Johannes Trapp

Thema der Arbeit

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

Stichworte

Fehlereinpflanzung, Mutationstests, NMEA-0183, GPS, virtueller Prototyp, Virtual Hardware in the Loop, Testszenarien.

Kurzzusammenfassung

Ziel dieser Arbeit ist die Durchführung eines Fault-Injection Experiments, anhand einer eigens dazu entwickelten Testumgebung. Die Implementation erfolgt nach dem Konzept der ”Virtual Hardware in the Loop”, unter Verwendung der Programmiersprache Java. Anhand zweier Testszenarien werden drei Fehlertypen abgeleitet, welche im Zuge des Experiments in den virtuellen Prototyp eingepflanzt werden. Ein weiterer Teil dieser Arbeit beschäftigt sich mit dem Thema der Automatisierbarkeit von Mutationstests sowie der Evaluation der entstandenen Testsuite durch den Einsatz von automatisierten Mutationstest-Tools.

Benjamin-Yves Johannes Trapp

Title of the paper

A concept for the test case development of safety-critical requirements by using fault injection and mutation tests

Keywords

Fault injection, Mutation test, NMEA-0183, GPS, virtual prototype, Virtual Hardware in the Loop, test scenarios.

Abstract

The aim of this thesis, is the performance of a Fault-Injection experiment based on a test Environment that was developed for this purpose. The implementation takes place according to the concept of the ”Virtual Hardware in the Loop” by using the programming language Java. Based on two test scenarios, three types of faults can be derived and injected into the virtual prototype. Another part of this thesis, engages with the topic of the automatation of mutation tests and the evaluation of the developed test suite by the use of automatic mutation test tools.

Inhaltsverzeichnis

1 Einleitung

Eingebettete Systeme sind aus unserem Alltag nicht mehr wegzudenken. Im Jahr 2010 erzielten sie in Deutschland ein Marktvolumen von über 19 Milliarden Euro[Sch10a]. Zunehmend finden sie als zentrale Recheneinheit an vielen Stellen in unserem Leben Einzug. Dazu gehören Smartphones, Navigationsgeräte und diverse industrielle Bereiche. Sie übernehmen Steuer- und Regelaufgaben jeglicher Art, verarbeiten Nutzereingaben und bereiten Informationen für den Anwender auf.

Viele dieser Mikroprozessorsysteme werden auch im Kontext eines sicherheitskritischen Systems entworfen, wie beispielsweise in der Medizintechnik. Bei einem sicherheitskritischen System handelt es sich um ein System, bei dem ein Fehler dazu führen kann, dass das Leben von Menschen oder die Umwelt gefährdet wird. Alltägliche Gegenstände wie eine Kaffeepadmaschine können bereits als sicherheitskritisches System eingestuft werden. Es genügt das ein Softwarefehler beim Erreichen des Siedepunktes eine Abschaltung des Gerätes verhindert. Als Folge könnte die Kaffeepadmaschine aufgrund eines Siedeverzugs[1] explodieren und Personen lebensgefährlich verbrühen.

Um die Eintrittswahrscheinlichkeit solcher Softwarefehler im Vorfeld zu minimieren, wird durch Einsatz von Softwaretests versucht, diese aufzudecken. Softwaretests erfolgen je nach Entwicklungsmodell analog zu den jeweiligen Entwicklungsphasen der Software. Es ist Standard das am Beispiel des V-Modells eine Validierung der implementierten Komponenten auf der Softwarearchitekturebene durch die Verwendung von Komponententests stattfindet. Diese Komponententests werden durch den Einsatz von häufig verwendeten Werkzeugen wie JUnit/TestNG für Java oder CPPUnit für C++ realisiert.

Die steigende Verbreitung und Komplexität der eingebetteten Systeme erfordern neben neuen Modellierungstechniken auch geänderte Entwurfsabläufe. Diese Änderungen betreffen neben der Entwicklung auch die Qualitätssicherung und führen unweigerlich zu einem Wandel der Testmethoden. Trotz sorgfältiger Durchführung von Softwaretests, muss mit dem Eintritt von

[1] Bezeichnung für ein Phänomen, bei dem Flüssigkeiten unter bestimmten Bedingungen über ihren Siedepunkt hinaus erhitzen, ohne das diese sieden

unvorhergesehenen Störungen im Betriebsablauf aufgrund latenter Softwarefehler gerechnet werden.

Ein Weg um einen gewissen Grad an Sicherheit zu erhalten und die Auswirkung unbekannter Seiteneffekte durch Softwaredefekte zu minimieren, führt zum Konzept der Fault-Injection. Bei diesem Konzept werden gezielt Fehler in ein System eingepflanzt, um das System auf Robustheit zu überprüfen. Diese Testmethode ist notwendig, da im Kontext der Zertifizierung von Sicherheitskritischen Systemen nach IEC61508, der Einsatz von herkömmlichen Testmethoden wie beispielsweise Komponententests nicht ausreichend sind. Im Zuge dieser Arbeit wird ein virtueller Prototyp erstellt, der als Pionierarbeit einen Teil des Systementwurfs in Form von Software abstrahiert. Dieser virtuelle Prototyp wird für die Durchführung eines Fault-Injection Experiments umfunktioniert. Ferner ist das Experiment an zwei Testszenarien gebunden. Anhand der beiden Testszenarien werden drei typische Fehlervarianten abgeleitet und im Rahmen des Fault-Injection Experiments in den virtuellen Prototypen eingepflanzt. Als Vertreter der betrachteten Fehlervarianten, spielen insbesondere Haftfehler für Hardwarebeschreibungssprachen eine wichtige Rolle. Um diese Thematik besser zu durchdringen, werden unabhängig vom virtuellen Prototyp, einige Konzepte zur Modellierung von Haftfehlern in VHDL vorgeführt.

Der Kern dieser Arbeit beschäftigt sich neben der Fault-Injection auch mit dem Einsatz von Mutationstests. Bei einem Mutationstest wird nicht das Softwaresystem, sondern die Qualität der dazugehörigen Komponententests überprüft. Das lateinische Sprichwort Quis custodiet ipsos custodes? (dt. Wer bewacht die Wächter) beschreibt die Semantik von Mutationstests formal am besten. Neben einer Analyse der Gründe für eine Automatisierbarkeit von Mutationstests, erfolgt im Rahmen dieser Arbeit eine Bewertung dreier automatisierter Mutationstest-Tools.

1.1 Randbedingungen dieser Arbeit

Diese Arbeit ist angelehnt an die Arbeiten des Projekts »Airborne Embedded Systems (AES)« des Departments Technik und Informatik der Hamburger Hochschule für Angewandte Wissenschaften (HAW). Dieses Projekt läuft im Zusammenhang mit dem studentischen Projekt BWB AC 20.30[2] des Departments Fahrzeugtechnik und Flugzeugbau der HAW, das sich zum Ziel gesetzt hat, einen Blended Wing Body (BWB) zu entwickeln. Die Projektgruppe AES verfolgt die Absicht, den BWB AC 20.30 als Flugdrohne umzufunktionieren.

Abbildung 1.1: Aussehen des AC20.30 BWBs (Quelle: www.ac2030.de)

In der durch das AES-Projekt zu entwickelnden Systemarchitektur der Flugdrohne kann die Flight Control Unit (FCU) als zentrale Hauptkomponente angesehen werden. Die FCU unterstützt den Piloten während des Fluges, erfasst Messwerte von diversen Sensoren und loggt diese auf einer externen SD-Karte. Wichtige Messdaten der Sensorik werden zwecks Lokalisierung der Messung direkt via Telemetrie zur Basisstation des Piloten gesendet. Zudem kommuniziert die FCU neben dem RC[3]-Empfänger, mit dessen Hilfe der Pilot die Flugdrohne steuert, direkt mit einem Field Programmable Gate Array (FPGA). Der FPGA manipuliert mittels Pulsweitenmodulation (PWM) die Servomotoren des Leitwerks der Flugdrohne. Die Nachfolgende Abbildung 1.2 zeigt ein SysML-Diagramm, das den eben skizzierten Sachverhalt detaillierter veranschaulicht.

[2] Quelle: http://www.ac2030.de/about/
[3] RC = Remote Control (dt. Fernsteuerung)

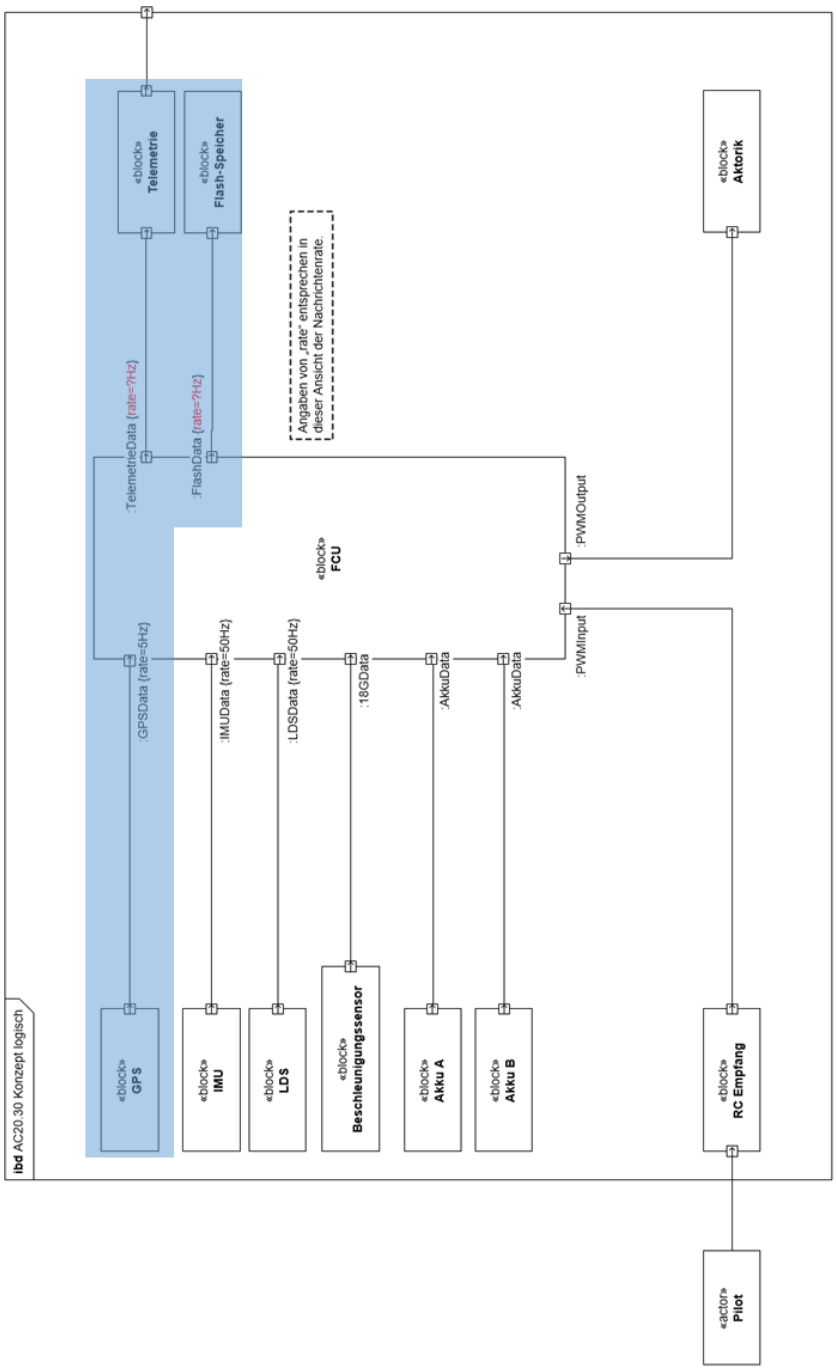


Abbildung 1.2: Ausschnitt aus dem SysML-Diagramm von René Büscher. Die blau markierte Stelle bildet die Grundlage für die Entwicklung des virtuellen Prototyps

Aktuell befindet sich das AES-Projekt in einem sehr frühen Entwicklungsstadium. Bis zum Zeitpunkt dieser Arbeit erfolgte lediglich eine Erhebung der Anforderungen für die zu erstellende Systemarchitektur sowie deren Modellierung in SysML. Daher kann im Rahmen dieser Arbeit nicht auf Codefragmente des AES-Projektes zurückgegriffen werden. Jedoch ist es für die Durchführung eines Fault-Injection Experiments und die Erprobung diverser Tools für den Mutationstest notwendig, eine eigens für dieses Ziel maßgeschneiderte Testumgebung zu implementieren.

Der in dieser Arbeit entwickelte virtuelle Prototyp wird zu einer Testumgebung umfunktioniert und stellt damit nicht nur eine Grundlage für weitere Projekte dar, sondern kann auch als erste Pionierarbeit angesehen werden. Um keine rein abstrahierte Testumgebung zu erstellen, verfolgt diese Arbeit das Konzept der Virtual Hardware in the Loop (siehe Kapitel 2.1.2). Da die Entwicklung der Testumgebung von sekundärer Bedeutung ist, wurde der Umfang zum Aufzeigen des Konzepts auf das GPS-Modul beschränkt. Die für die Entwicklung des virtuellen Prototyps notwendigen Komponenten sind in dem SysML-Diagramm (Abbildung 1.2) durch eine blaue Markierung hervorgehoben. Die daraus resultierende Architekturbeschreibung dient als Grundlage der Implementierung des Prototyp und wird in Kapitel 3.1 explizit diskutiert. Die Umwandlung des virtuellen Prototyps in eine Testumgebung, wird in Abschnitt 3.2.4 und 3.2.5 erörtert.

Bedingt durch das Konzept der vHIL, stellt der zu implementierende virtuelle Prototyp nicht nur ein Abbild eines kleineren Teils der geplanten Systemarchitektur dar, sondern eine möglichst realistische Abbildung von Hardware. Dieses Konzept verfolgt mehrere Ziele parallel, so soll möglichst frühzeitig die Option geboten werden, Fragen und Probleme begleitend zur Entwicklung der FCU-Systemarchitektur zu klären. Aber auch die Entwicklung und Simulation der Hardware soll unter realistischeren Bedingungen ermöglicht werden. Darüber hinaus soll der virtuelle Prototyp das Entwicklungsteam in die Lage versetzen, noch vor der Fertigstellung der realen Hardware mit der Qualitätssicherung zu beginnen.

Hierbei soll die Qualitätssicherung nicht nur die Erstellung von zuverlässigen Tests umfassen, sondern auch Konzepte aufzeigen, die bei der Entwicklung von Mechanismen signifikant beitragen, um eine Immunisierung der Soft- und Hardware gegenüber vordefinierten Fehlertypen zu erlangen. Aber auch die Qualität der Testsuite des virtuellen Prototyps kann durch den Einsatz diverser automatisierter Mutationstest-Tools gesteigert werden, um ein höheres Vertrauen in ihre Effektivität zu gewinnen.

2 Grundlagen

2.1 Grundlegende Begriffe

Um einen leichteren Einstieg und Verständnis für die Kernthemen dieser Arbeit zu ermöglichen, werden zunächst die grundlegenden Begriffe eingeführt. In diesem Abschnitt wird zudem auf »Virtual Hardware in the Loop (vHIL)« eingegangen. Dieser Begriff steht primär im Kontext der zu entwickelnden Testumgebung.

Eng mit dem Kontext verflochten sind die Normen IEC 61508 und ISO/FDIS 26262. Daher wird auf an den entsprechenden Stellen Bezug auf diese Normen genommen.

2.1.1 Sicherheitskritische Anwendungen

Diese Arbeit betrachtet sicherheitskritische Anwendungen im Sinne von Software. Die DIN Norm IEC 61508 Teil 3 behandelt die Sicherheitskritikalität von Software. Da Software im Gegensatz zur Hardware frei von physikalischen Einflüssen ist, kann sie nicht ausfallen. Ein Softwaresystem kann jedoch aufgrund von Fehlern innerhalb der Software selbst zu Fehlfunktionen führen. Durch Softwarefehler wird das Softwaresystem mit einem falschen Wert oder einer falschen Steueranweisung betrieben. Dies äußert sich im schlimmsten Fall in einer permanenten Unterbrechung des Programmablaufs und kann Auswirkungen auf das gesamte System haben [Som07].

Daraus lässt sich schlussfolgern: Wenn ein System durch einen Softwarefehler in einen gefährlichen Zustand versetzt werden kann, ist die Software als sicherheitskritisch anzusehen. Dies gilt insbesondere, wenn sich die Anforderung nach Sicherheit aus der Anforderung nach Fehlerfreiheit ableitet [Sto96].

2.1.2 Virtual Hardware in the Loop (vHIL)

In seinem Whitepaper »Virtual Hardware „In-the-Loop“: Earlier Testing for Automotive Applications« [Rey13] beschreibt der Autor Victor Reyes das Konzept der vHIL. Die vHIL verfolgt das Ziel, den Prozess des Testens in der Entwicklung noch weiter nach vorne zu verlagern. Um dies zu erreichen, werden sogenannte virtuelle Prototypen eingesetzt. Bei einem virtuellen Prototyp handelt es sich um ein Softwaremodell, das die zu erstellende reale Hardware emuliert. Typischerweise kann laut Autor das Modell auf einem herkömmlichen Desktop-PC simuliert werden. Ein großer Vorteil dieser virtuellen Prototypen ist laut Reyes die Möglichkeit, die identische binäre Software auszuführen die später auf der realen Hardware läuft, ohne das Modell zu verändern.

Dies soll Softwareteams in die Lage versetzen, anhand eines realistischen Abbilds der Hardware, die im Rahmen der Qualitätssicherung entstehenden Softwaretests bereits vor der Fertigstellung der realen Hardware durchzuführen. Durch diese Frontverlagerung des Testprozesses sollen laut Autor, neben einer besseren Qualität der Tests auch eine höhere Qualität der Software und letztendlich ein geradliniger „in-the-Loop“-Fluss entstehen.

2.1.3 Fault-Injection

Die ISO/FDIS 26262 sowie die DIN Norm IEC 61508 Teil 7, C.5.6 beschreiben den Software Fault-Injection Test als eine relevante Methode zum Testen funktionaler Sicherheitsstandards. Die Fault-Injection selbst ist eine Technik um die Testabdeckung zu verbessern. Dies geschieht durch Hinzufügen diverser Fehler an unterschiedlichen Bereiche des Softwaresystems. Hierbei ist vor allem die Fehlerbehandlung von besonderer Bedeutung um die Robustheit des Softwaresystems sicher zu stellen. Folglich ist die Fault-Injection als ein Robustheitstest zu verstehen, mit dem Ziel schwer test- oder reproduzierbare Schwachstellen von Software zu überprüfen [VM98]. Die Fault-Injection umfasst neben Soft- und Hardware auch diverse Hybridformen. Neben Kapitel 4 befinden sich im Anhang an diese Arbeit weitere Informationen, praktische Beispiele, und detailliertere Erläuterungen.

2.1.4 Mutationstest

Der Mutationstest wurde 1978 von DeMillo, Lipton und Sayward [RLS78] vorgeschlagen. Ein Mutationstest ist ein Test, bei dem die Originalversion der zu testenden Software verändert wird um die Qualität der Testfälle mit Hilfe dieser Mutanten zu bewerten. Bei einem Mutanten handelt es sich um eine Softwareversion, die unter Anwendung einer Mutations-Transformation erzeugt wird und genau einen bekannten Fehler enthält[VM98]. Die Mutationstransformation ist eine vordefinierte Regel, bei der eine einzelne Anweisung zu einem bestimmten Programmfehler führt. Um dies besser zu veranschaulichen, dient das nachfolgende Listing:

```
1 \\Original If-Verzweigung
2 if(x > y){
3 ...
4 }
5
6 \\Mutant
7 if(x >= y){
8 ...
9 }
```

Listing 2.1: Exemplarische Mutation einer If-Verzweigung

Zusammengefasst lässt sich der Mutationstest als eine fehlerbasierende Testmethode kategorisieren. Sie verlangt keine geänderte Softwareerstellung, sondern leitet diversitäre Softwarevariationen aus der Originalversion ab. Der Mutationstest testet also nicht das Programm, sondern die Tests mit dem Ziel, die optimalen Testdaten zu bestimmen [Bud80].

2.2 Fault-Injection Environment

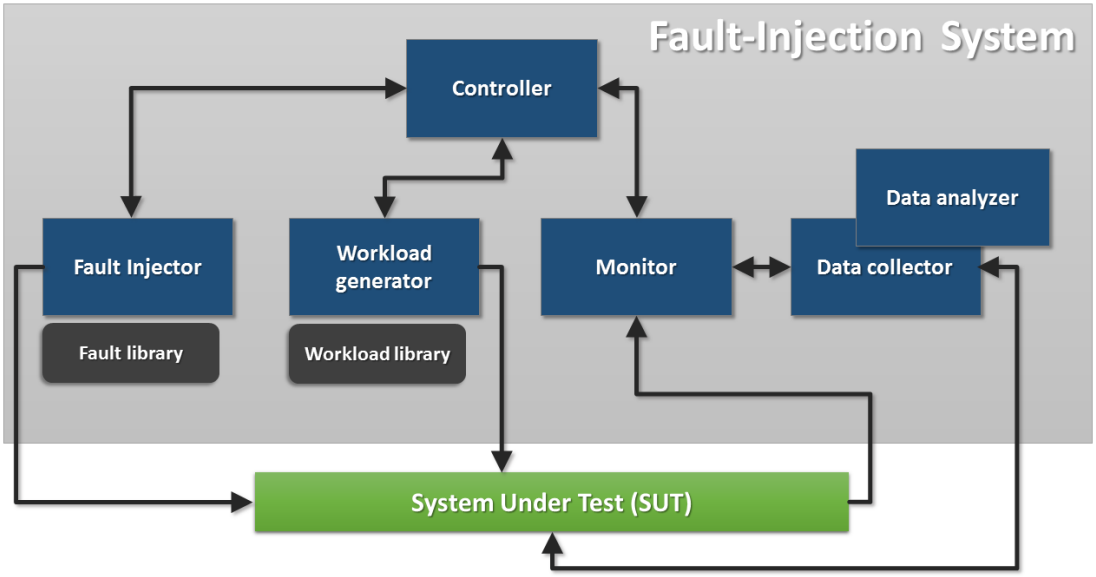


Abbildung 2.1: In Anlehnung an [HTI97, S.76]. Basis Komponenten und deren Aufbau zu einer Fault-Injection Environment

Die oben dargestellte Abbildung 2.1 entstand in Anlehnung an der im Artikel [HTI97] vorgestellten Basis Fault-Injection Environment. Dabei besteht die Fault-Injection Environment aus dem Zielsystem (SUT - System Under Test), mit dem das Fault-Injection System zwecks Durchführung eines Fault-Injection Experiments kommunizieren soll. Das Fault-Injection System selbst besteht nach [HTI97] aus den nachfolgenden Basiskomponenten:

- Der Fault injector injiziert die Fehler in das Zielsystem und führt alle Kommandos des Workload generator aus. Des weiteren unterstützt der Fault injector eine Vielzahl an verschiedenen Fehlertypen, -orten, -zeiten und zweckmäßigen Hard- oder Software Strukturen die in der Fault library hinterlegt sind. Die Fault library selbst ist eine separate Komponente die eine größere Flexibilität und Portierbarkeit bereitstellen soll.
- Der Workload generator erzeugt die Ausführungskommandos die durch das SUT verarbeitet werden sollen. Hier können mit Hilfe der Workload library diverse reguläre Szenarien für den Regelbetrieb des SUTs verwendet werden.
- Der Monitor überwacht die Ausführung der Kommandos und leitet bei Bedarf die Sammlung der Daten ein.

- Mit Hilfe des Data collectors werden die Daten (meist online) gesammelt.
- Durch den Data analyzer (in der Regel offline) werden die Daten verarbeitet und analysiert.
- Der Controller steuert das Fault-Injection Experiment. Der Controller selbst ist ein Programm, das entweder auf dem Zielsystem oder verteilt auf einem remote Computer operiert.

Diese Zusammenstellung bildet die Grundlage für Fault-Injection Experimente und lässt sich in der einen oder anderen Form in diversen Tools wie ByteMan [Byt] oder Xception [Xce] für die Hard- bzw. Software Fault-Injection und deren Hybridformen wiederfinden. Aufgrund dieser Bedeutsamkeit wird das in dieser Arbeit vorgestellte Fault-Injection Experiment (siehe Kapitel 3.2.5) in starker Analogie zu obiger Basis Fault-Injection Environment gestaltet.

2.3 NMEA-0183 Datensätze

Als real existierender GPS[1]-Empfänger wird für die Flugdrohne AC20.30 ein GPS Empfänger aus der GPS18x Serie der Firma GARMIN verwendet. Die empfangenen GPS-Daten werden in Form von NMEA-0183 Sätze empfangen.

Bei der National Marine Electronics Association (NMEA) handelt es sich um eine US-amerikanische Vereinigung von Elektronikherstellern und -Händlern der Schifffahrtsindustrie [Nmeb]. Die NMEA definierte den NMEA-0183 Standard für die Kommunikation zwischen Navigationsgeräten auf Schiffen mittels GPS-Endgeräten. Der Standard selbst besteht aus einer RS422-Schnittstelle und standardisierten Datensätzen.

Für die Testumgebung spielen die beiden NMEA-0183 Sätzen GGA und RMC auf Grund ihrer Semantik eine maßgebende Rolle. Um die kryptischen GPS-Daten und deren Bedeutung für die Flugdrohne besser verstehen zu können, wird der generelle Aufbau von NMEA-Datensätzen und beiden Sätze GGA und RMC nachfolgend explizit diskutiert.

[1] Global Positioning System (GPS) - Globales Navigationssatellitensystem zur Positionsbestimmung und Zeitmessung

Der formelle Aufbau eines NMEA-Datensatzes für GPS genügt in der Regel der nachfolgend dargestellten allgemeinen Form und umfasst eine maximale Länge von 80 Zeichen [Gar]:

$GP<- - ->,<X>,...,<X$_n$>*<Checksum><CR><LF>

- *$GP* – Talker ID hierbei steht GP für GPS, GL für GLONASS[2]
- <- - -> – Name des Datensatzes z.B. RMC, GGA, DTM
- *<X>* – Vordefinierte Menge an Daten die im Kontext des Datensatzes stehen
- *<Checksum>* – Berechnung der Checksumme anhand Bildung der Kontrajunktion (XOR) des gesamten NMEA-Satzes
- *<CR><LF>* – Zeilenumbruch als Terminierungszeichen (CR = Carriage Return, LF = Line feed)

Jeder GNSS[3]-Empfänger muss als Minimum mindestens den Recommended Minimum Sentence C (RMC) Datensatz unterstützen und diesen einmal alle zwei Sekunden ausgeben können. Dieser Datensatz enthält alle notwendigen Informationen um die Geolokalisierung des Empfängers durchzuführen. Daher ist es unerlässlich, dass auch der virtuelle Prototyp diesen Datensatz unterstützt.

Da der RMC-Satz keine Angabe zur Höhe enthält, diese jedoch für die Flugdrohne essentiell ist, bietet sich die Verwendung des Global Positioning System Fix Data (GGA) Satzes an. Der GGA Satz enthält als einziger NMEA-Datensatz die notwendigen Daten für die Bestimmung der aktuellen Höhe des Empfängers über dem Meeresspiegel. Die Häufigkeit der Ausgabe des Satzes beträgt üblicherweise 1Hz (sprich eine Ausgabe pro Sekunde). Die nachfolgenden Tabellen 2.1 und 2.2, geben einen genaueren Überblick über die Struktur der beiden NMEA-Sätze.

[2] Globalnaya navigatsionnaya sputnikovaya sistema (GLONASS) - Russisches Pendant zum amerikanischen GPS
[3] Globales Navigationssatellitensystem

	$GPRMC,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,<9>,<10>,<11>,<12>*<Checksum><CR><LF>
<1>	UTC Time (hhmmss.ss)
<2>	Status der Bestimmung: A=Active (gültig); V=Void (ungültig), P = Precise
<3>	Breitengrad
<4>	(N=Nord, S=Süd)
<5>	Längengrad)
<6>	(E=Ost, W=West
<7>	Geschwindigkeit über Grund in Knoten
<8>	Bewegungsrichtung in Grad
<9>	Datum (ddmmy)
<10>	Magnetische Deklination von 000.0 bis 180.0 in Grad
<11>	Richtung der Magnetischen Deklination (E=Ost, W=West)
<12>	Modus A=Autonom, D=Differential, E=Estimated, N=Data not valid S=Simulated

Tabelle 2.1: Erläuterung des RMC Datensatz. Entnommen aus den Technischen Spezifikationen des Garmin GPS18x [Gar]

	$GPRMC,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,<9>,<10>,<11>,<12>,<13>,<14>*<Checksum><CR><LF>
<1>	UTC Time (hhmmss.ss)
<2>	Breitengrad
<3>	(N=Nord, S=Süd)
<4>	Längengrad)
<5>	(E=Ost, W=West
<6>	GPS-Qualitäts Indikator 0= fix Nicht verfügbar, 1=GPS fix, 2=Differential GPS fix
<7>	Anzahl an Satelliten in Reichweite (00 bis 12)
<8>	Horizontal Dilution of Precision
<9>	Höhe der Antenne über dem Meeresspiegel
<10>	M = Höhenangabe in Metern
<11>	Höhe Geoid minus Höhe Ellipsoid (WGS84)
<12>	M = Höhenangabe in Metern
<13>	Alter der Differential GPS Daten in Sekunden, seit letztem Refresh (0, ohne DGPS)
<14>	ID der Differential GPS referenz Station, 0000-1023

Tabelle 2.2: Erläuterung des GGA Datensatz. Entnommen aus den Technischen Spezifikationen des Garmin GPS18x [Gar]

2.4 NMEA-Checksumme

Da die NMEA-Checksumme die zentrale Methode zur Erkennung von Invarianzen in NMEA-Datensätzen darstellt, lohnt sich ein genauer Blick auf das dazu verwendete Berechnungsverfahren.Das Verfahren zur Berechnung der Checksumme ist relativ trivial. Hierbei wird die Checksumme selbst durch zwei Hexadezimalzahlen repräsentiert und durch Verwendung der (logischen) XOR-Operation über alle Zeichen des NMEA-Datensatzes gebildet, die zwischen den Zeichen '$' und '*' stehen. [Rie08].

Anhand des eben formal beschriebenen Algorithmus, entstand der nachfolgenden Java-Code:

```
1  public String createNMEAChecksum(String nmeaSentence) {
2          if(nmeaSentence == null || nmeaSentence.equals(""))
3                  throw new NullPointerException();
4
5          //Satz muss mit einem $-Zeichen beginnen
6          if(!nmeaSentence.contains("$"))
7                  throw new RuntimeException();
8
9          //Satz hat u.U. schon eine Checksumme
10         if(nmeaSentence.contains("*"))
11                 throw new RuntimeException();
12
13         int chk = 0;
14         String chk_s = "";
15
16         //i == 1 um das '$' Zeichen zu überspringen
17         for (int i = 1; i < nmeaSentence.length(); i++)
18                 chk ^= nmeaSentence.charAt(i);
19
20         chk_s = Integer.toHexString(chk).toUpperCase();
21
22         if(chk_s.length() < 2)
23                 chk_s = "0" + chk_s;
24
25         return chk_s;
26 }
```

Listing 2.2: Java Implementierung der NMEA-Checksummen Berechnung

3 Anforderungsanalyse und Implementierung

Zur Entwicklung eines virtuellen Prototyps für die Testumgebung, dient im Rahmen dieser Bachelorarbeit, das GPS-Modul der Flugdrohne AC20.30 als Vorbild. Die Geolokalisation der Flugdrohne erfolgt mittels GPS-Daten. Die GPS-Komponente stellt die Achillesverse im Kontext des autonomen Flugs der Flugdrohne dar. Deshalb wird die Semantik des autonomen Flugs und die Schwachstelle GPS-System nachfolgend diskutiert.

Arne Richter beschreibt in seiner Safety-Analyse [Ric13] die Anforderungen für einen autonomen Flug, als einen Flugmodus, bei dem die Drohne zu einer konkreten Position im Raum navigiert und kein Vektor mehr vorgegeben werden muss. Der Pilot übermittelt hierbei die Zielposition per GPS-Daten an die Flugdrohne. Die Flugdrohne ermittelt nun selbstständig aus ihrer Ausgangs- und der empfangenen Zielposition einen Vektor im Raum, den sie anschließend abfliegt.

Darüber hinaus soll ein Teil der dokumentierten Messdaten mit GPS-Positionsangaben versehen werden. Solche Messdaten werden beispielsweise durch den Luftdaten Sammler (LDS) erhoben. Die Geolokalisierung via GPS dient hierbei zur Dokumentation der Position, an der die Messung durchgeführt wurde.

Ein autonomer Flug birgt jedoch auch Risiken in sich. In seinem Artikel [Möc12] für den Österreichischen Rundfunk ORF beleuchtet der Autor Erich Möchel kritisch, wie Störungen im GPS-System zum Absturz einer österreichischen Flugdrohne führte. In der europäischen Luftfahrt ist das GPS als einzige Navigationsquelle für den nicht zivilen Flugverkehr unzulässig [Klo12]. Der Einsatz der zivilen Variante des GPS ist im Privat- und Hobbysektor üblich und für die meisten Anwendungsgebiete ausreichend. Jedoch ist die Anfälligkeit für eine Störung durch einen GPS-Störsender beim Einsatz von GPS als einzige Navigationsquelle sehr wahrscheinlich, wie der Absturz der österreichischen Flugdrohne aufzeigt. So kann mittels Störsender das GPS-Modul sowohl gestört als auch manipuliert werden. Im Internet lassen sich diverse Quellen mit Bauanleitungen für einen kostengünstigen GPS-Störsender auffinden [Mag07].

Aufgrund dieser Störanfälligkeit des zivilen GPS-Systems und dessen Bedeutung für die Flugdrohne bietet es sich an, das GPS-Modul in einer abstrahierten Form als virtuellen Prototyp zu realisieren. Dieser virtuelle Prototyp bietet eine Vielzahl an Einsatzmöglichkeiten und zudem eine attraktive Grundlage für die Durchführung eines Fault-Injection Experiments.

3.1 Architektur des virtuellen GPS-Modul-Prototyps

Da die gesamte Systemarchitektur diverse Komponenten zur Verfügung stellt, wird als Vorbild für den zu entwickelnden virtuellen Prototyp, analog zu den Randbedingungen (siehe Kapitel 1.1) die Entwicklung lediglich auf das GPS-Modul beschränkt. Hierbei dient die blau markierte Stelle des SysML-Diagramms der Abbildung 1.2 als Vorbild. Anhand dieses Vorbilds, wurde das UML-Komponentendiagramm in Abbildung 3.1 modelliert.

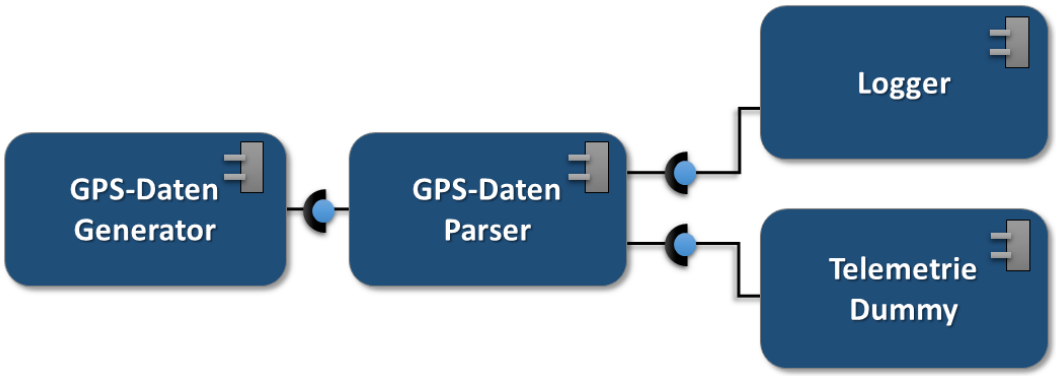


Abbildung 3.1: Komponentendiagramm des GPS-Modul Prototyps in Anlehnung an das SysML-Diagramm von René Büscher

Da im Kontext dieser Arbeit nicht der virtuellen Prototypen im Vordergrund steht, sondern das Fault-Injection Experiment und die Erprobung von Mutationstests anhand der entstehenden Testsuite dieser Architektur, wird an dieser Stelle auf eine detaillierte Beschreibung der einzelnen Komponenten mittels UML-Klassendiagramm verzichtet.

Statt UML-Diagramme einzusetzen, erfolgt die semantische Erläuterung der einzelnen Komponenten in Prosa. Diese Variante wurde primär gewählt, um die Hintergründe und deren Funktionalität zu durchleuchten und gleichzeitig den Bezug auf die Kernthemen dieser Arbeit zu verstärken.

3.1.1 GPS-Generator

Die Klassen TimerTask und Timer dienen zur Implementierung von zeitgesteuerten Abläufen und gehören zur Java-Biliothek java.util. Die Klasse Timer dient zur zeitgesteuerten Ausführung einer Aufgabe, die mittels TimerTask formuliert und zu einem festen Zeitpunkt periodisch oder einmalig durch den Timer ausgeführt wird. Beide Klassen werden zur Erstellung von pseudo-randomisierten Daten verwendet, die zur Erstellung von NMEA-Datensätzen essentiell sind.

Die GPS-Generator Komponente unterstützt hierbei drei Varianten. Die randomisierten Zahlen werden für die jeweiligen Basisdaten des NMEA-Satzes, wie beispielsweise der Längen- oder Breitengrad, einmal alle 0,5 Sekunden separat in Abhängigkeit ihres gewählten Modus für die Randomisierung aktualisiert. Die Formatierung der Zahl erfolgt spezifisch ihrer Darstellung im NMEA-Datensatz, da diverse Unterschiede bei der Anzahl der Nachkommastellen vorherrschen. Für die Randomisierung der einzelnen NMEA-Basisdaten stehen die folgenden Modi zur Verfügung:

- ASCENDING - Dieser Modus addiert eine generierte Zufallszahl mit dem aktuellen Basisdatum der Positionsangaben und sorgt dadurch für eine aufsteigend Tendenz bei der Randomisierung der aktuellen Position.
- DESCENDING - Analog zum ASCENDING Modus wird hier die generierte Zufallszahl vom gewählten Basisdatum subtrahiert, für eine absteigende Tendenz der generierten Positionsangaben.
- RANDOM- Hierbei wird die generierte Zufallszahl auf ihre Parität überprüft. Entsprechend ihrer Parität wird die anschließende Zahl in- oder dekrementiert. Dadurch soll eine von einer auf- oder absteigenden Tendenz losgelöste Positionsangabe ermöglicht werden, die komplett dem Zufall überlassen ist.

Nach dem Randomisieren der NMEA-Basisdaten werden durch zwei weiteren TimerTasks, die beiden NMEA-Sätze GGA und RMC generiert und mittels Stream (Bestandteil des Kommunikationsmoduls) an die GPS-Parser Komponente versendet.

3.1.2 GPS-Parser

Anhand der Beschreibung der beiden NMEA-Sätze aus Kapitel 2.3 wird deutlich, dass unter anderem wegen des kryptischen Aufbaus, die Daten so direkt nicht durch Software verwertbar sind. Um aus den übertragenen NMEA-Datensätzen, ein für die Weiterverarbeitung durch Software verwertbares Format zu konvertieren, ist es notwendig, einen Parser zu implementieren. Jeder Parser muss für die durchzuführende syntaktische Analyse einer Grammatik folgen. Die vom implementierten Parser verwendete Grammatik ist im nachfolgenden Listing 3.1 dargestellt:

```
1  <NMEASentence> ::= <Key><Delimiter><Data><Delimiter><EndOfSentence>
2  <Key>          ::= ?\$GPRMC?|?$GPGGA?
3  <Data>         ::= <Value>|<Value><Separator><Data>
4  <Value>        ::= <Char>|<Digit>{<Digit>}|<Digit>{<Digit>}.<Digit>{<Digit>}|<Empty>
5  <Char>         ::= a|A|b|B|c|C|d|D|e|E|f|F|N|n|S|s|W|w|$|*|<Empty>
6  <Digit>        ::= 0|1|2|3|4|5|6|7|8|9|<Empty>
7  <Delimiter>    ::= ,
8  <Empty>        ::= ?
9  <EndOfSentence>::= <Checksum><CR><CL>
10 <Checksum>     ::= *<Char>|<Digit><Char>|<Digit>
11 <CR>           ::= 0x0D
12 <CL>           ::= 0x0A
```

Listing 3.1: Kontextfreie Grammatik des implementierten GPS-Parsers in der Backus-Naur-Form

Die NMEA-Daten des GPS-Generators werden mittels Stream via Kommunikationsmodul empfangen und anschließend mit Hilfe des Parsers analysiert. Die durch den Parser analysierten Daten werden in einem extra Objekt gespeichert und an den Telemetrie Dummy weitergeleitet, um den Prozess der Verarbeitung zu simulieren.

3.1.3 Logger und Telemetrie Dummy

Um die Speicherung der Daten zu simulieren, wird die Simple Logging Facade for Java (SLF4J) verwendet. Die SLF4J stellt eine Java Logging API zur Verfügung, die zur vereinfachten Verwendung des Log4J-Loggers das Facade-Pattern einsetzt [Sch10b]. Der Logger wird unabhängig von der Speicherung der Daten auch zum generellen Monitoring des virtuellen Prototyps verwendet. Um jedoch nicht allein Log4J als Logging-Instrument zu verwenden, nutzt der Telemetrie Dummy die Java "Hauseigene" FileWriter-Klasse aus der java.io Standard-Bibliothek. Der Telemetrie Dummy schreibt mit Hilfe des FileWriters die durch den GPS-Parser gewonnenen Daten in eine eigene Log-Datei. Darüber hinaus ist der FileWriter im Gegensatz zum bereits bewährten und sehr gut getesteten Log4J-Logger nur so robust, wie der Entwickler es selbst definiert und implementiert. Diese Stelle bietet sich somit gut zum Einpflanzen von Fehlern an.

Als Ziel für das Logging mittels SLF4J resp. Log4J, werden neben der Java-Konsole alle relevanten Daten auch in eine Log-Datei geschrieben. Dies dient neben der Dokumentation auch für eine ggf. spätere Analyse des simulierten Flugs.

3.1.4 Kommunikationsmodul

Da im realen System die Daten des GPS-Empfängers mittels RS422 Schnittstelle an die FCU übermittelt werden, wird dies durch ein Kommunikationsmodul symbolisch im virtuellen Prototyp abgebildet. Das Kommunikationsmodul kapselt den benötigten In- und Output Bytestream sowie den BufferedReader aus der java.io Standard-Bibliothek von Java in ein allgemeingültiges Modul.

Dadurch erhält man unter anderem die Möglichkeit, die virtuell generierten NMEA-Datensätze mit Hilfe eines USB-zu-RS422 Konverters, an einem realen System zu testen. Darüber hinaus bietet das Kommunikationsmodul ein geeignetes Fundament zur Fault-Injection an, welches im Rahmen des Fault-Injection Experiments in Abschnitt 3.2.4 und 3.2.5 in einer variierten Form als Fault Injector verwendet wird.

3.2 Testszenario und Anforderungen

Die implementierte Software des virtuellen Prototyps dient zur Erprobung einiger Konzepte innerhalb eines Fault-Injection Experiments. Für dieses Experiment, wird die Software nach dem Vorbild der in Abschnitt 2.2 vorgestellten Fault-Injection Environment umfunktioniert. Dieser Vorgang wird im Abschnitt 3.2.4 und 3.2.5 genauer diskutiert.

Um einen realen Bezug zu erhalten, soll das Konzept verschiedene Fehlertypen unterstützen. Daher behandelt dieser Abschnitt zwei Szenarios und die daraus resultierenden Anforderungen, um für das Experiment einen Bezug zur Realität herzustellen. Diese Anforderungen bilden zudem, neben der Ableitung von Fehlertypen, die Randbedingungen des Experiments. Die drei anhand dieser Szenarios abgeleiteten Fehlertypen werden im Zuge ihrer Implementierung einer abstrahierten Form den realen Fehlertypen nachempfunden. Die Abstrahierung ist notwendig, da das Ziel dieser Testszenarien nicht das Auffinden von Fehlern ist sondern vielmehr die Funktion als Prüfkonzept. Anhand dieses Prüfkonzepts, kann ein Nachweis der Robustheit und Funktionalität der zu diesem Zweck implementierten Mechanismen gewährleistet werden. Neben einem Nachweis über die Robustheit, dient dieses Prüfkonzept auch für die Entwicklung neuer Mechanismen, die eine Immunisierung der Software gegenüber die eingepflanzten Fehlertypen ermöglichen.

Die für das Testszenario entwickelte Software ist so ausgelegt, dass sie lediglich in Kombination mit dem implementierten virtuellen Prototyp interagieren kann. Um die Software generisch zu gestalten, sind weitere evolutionäre Schritte innerhalb der Software notwendig. Neben der Erläuterung der beiden Szenarien und den drei daraus resultierenden Fehlertypen, wird in diesem Kapitel auch auf die Konzeption des Fault-Injection Experiments eingegangen.

3.2.1 Szenario - Kosmische Strahlung

Dieses Szenario bezieht sich auf die Auswirkungen von radioaktiver- und kosmischer Strahlung auf Hardware. Als Grundlage hierfür, dient die Diplomarbeit von Thomas Juhnke [Juh03]. Wohl jeder hat bereits davon gehört, dass radioaktive- und kosmische Strahlung Fehler in Rechnern verursachen kann, die in der Literatur als Soft-Errors bezeichnet werden. Die Bezeichnung Soft-Error kommt daher, weil nicht die Hardware selbst, sondern gespeicherte Informationen beeinträchtigt werden [Sof].

Im Bereich der radioaktiven Strahlung spielt besonders die α-Strahlung eine bedeutende Rolle im Zusammenhang mit Soft-Errors. So identifizierten May und Woods im Jahre 1979 α-Teilchen

als Ursache für Soft-Errors in dynamischen Speichern (DRAMs) [TW79]. Bei α-Teilchen handelt es sich um zweifach positiv geladene Heliumionen (Darstellung innerhalb der Physik und Chemie: HE^{2+}), die laut Juhnke durch radioaktiven Zerfall von nicht völlig vermeidbaren Uran- und Thorium-Spuren im IC-Gehäuse und der Aluminium-Metallisierung entstehen. Aber auch die galaktische kosmische Strahlung besteht neben 87% Protonen (Wasserstoffkerne) und 1% Myonen (schweren Atomkernen) aus ca. ∼12% Alpha-Teilchen [Dem10]. Mittlerweile gelten die α-Teilchen, die aus Verunreinigungen des Gehäusematerials stammen als beherrschbar [Sof]. Jedoch bleibt die Beeinträchtigung durch kosmische Strahlung weiterhin bestehen.

Ein α-Teilchen hinterlässt im Silizium eine Spur von Elektronen-Loch-Paaren entlang seiner Bahn. Dabei können laut May und Woods bei einer Energie des α-Teilchen von 5 MeV (Elektronenvolt) bis zu $1{,}4*10^6$ Elektronen-Loch-Paare erzeugt werden, wodurch es letztendlich zu einer Umladung von Schaltungsknoten kommt, und sich in Form eines Soft-Errors bemerkbar macht [Juh03].

Die Software muss laut Juhnke beim Schutz gegen Verfälschung in mindestens zwei Bereiche unterschieden werden. Juhnke stellt unter diesem Aspekt einige Methoden zur Sicherung von Datenbereichen in seiner Arbeit vor. Als besonders erwähnenswerte Hardwaremethode ist hierbei der Schutz vor Datenverfälschung im Prozessor selbst (sofern dieser keine eigene Fehlerkorrektur bietet) zu erwähnen. In Korrelation mit dem virtuellen Prototypen und dem Fault-Injection Experiment ist dies jedoch nicht ohne weiteres umsetzbar. Daher wird in diesem Szenario auf die Auswertung der vorhandenen Prüfsumme zurückgegriffen. Jeder NMEA-Datensatz verfügt zum Schutz vor Korrumpierung über eine Prüfsumme. Allerdings ist die Verwendung optional und muss nicht zwingend angegeben bzw. ausgewertet werden. Neben der Funktion als Prüf- und Schutzmechanismus erfolgt anhand dieses Szenarios zudem eine Analyse über den Zuwachs an Robustheit durch eine konsequente Auswertung der Prüfsumme. Hierfür wird eine eigene Klasse erstellt, die sowohl die Checksumme generieren als auch validieren kann um die Mechanismen in einem Punkt zu kapseln.

Unabhängig von einer Verfälschung der Daten können auch Haftfehler bedingt durch α-Teilchen auftreten. Ein Haftfehler basiert auf der Annahme, dass ein Eingang, Ausgang oder eine Leitung auf einem festen Wert liegt und dadurch keinerlei Signalwechsel möglich sind. Dies kann durch Materialfehler ausgelöst werden oder aber durch eine Korrumpierung der Software bzw. der Daten mit denen die Software arbeitet [HM04]. Wenn ein Datum innerhalb der Software bei einem Zustandswechsel verfälscht wird, kann aufgrund der ungültigen Transition die Software in einen ungültigen Zustand gelangen. Durch das Verweilen in diesem ungültigen

Zustand ist ein Haftfehler als Resultat von außen beobachtbar [VM98]. Als Auslöser für einen Haftfehler wäre daher auch eine Korrumpierung durch α-Teilchen beim Zustandswechsel der Software denkbar. Da ein Haftfehler eine besondere Bedeutung für Hardwarebeschreibungssprachen wie VHDL oder Verilog hat, erfolgt eine separate Behandlung dieses Themas als Exkurs in Kapitel 4.

Neben α-Teilchen, führen auch elektromagnetische Entladungen oder Kopplungen zu Haftfehlern. Infolgedessen, können sporadisch auftretende Abweichungen der Spannungspegel auftreten, die in Transistoren Verzögerungen bei der Signalverarbeitung hervorrufen. In diesem Zusammenhang spricht man dann von sogenannten overshoots im Falle einer Überspannung und analog bei Spannungseinbrüchen von glitches[HM04]. Eine realistische Abbildung eines solchen Szenarios, würde jedoch den Rahmen sowie die Möglichkeiten von Java überstrapazieren. Daher empfiehlt es sich, ein Fault-Injection Experiment sowie eine analytische Beobachtung von glitches und overshoots unter Verwendung der Hardwarebeschreibungssprache VHDL, in einer separaten Arbeit zu untersuchen.

Für weiterführende Informationen zu diesen beiden Fehlertypen- respektive Modelle, findet im Kapitel 7.2 des Anhangs, eine generelle Erläuterung diverser HWFI Techniken statt. Darunter wird auch auf die strahlenbasierte Injektion eingegangen, die stark mit diesem Testszenarios verbunden ist. Ferner werden dort auch Techniken erläutert zum Auffinden von (oftmals fertigungsbedingten) Materialfehler in elektrischen Schaltungen. Diese Techniken zeigen im Rahmen ihres realen Auftretens sowie in der Simulation durch Fault-Injection oftmals auch ein für Haftfehler typisches Verhalten.

3.2.2 Szenario - GPS-Spoofing

Neben den eben aufgeführten Fehlern aus dem ersten Szenario ist auch das Auftreten von Sprüngen in den empfangenen Daten sehr wahrscheinlich. Laut einem Homelandsecurity Whitepaper [WJ03] sind Sprünge in der Angabe des Längen- und Breitengrads typisch für GPS-Spoofing Angriffe. Hierbei werden vom Angreifer ausgesendete Störsignale verwendet, um regulär empfangene GPS-Signale des Zieles zu imitieren. Diese Imitation verfolgt das Ziel den Empfänger des GPS-Signals mit falschen Positionsangaben zu täuschen [WJ03].

Ein solches Szenario ist aufgrund der geringen Kosten von ca. 1000 Dollar für einen GPS-Signalfälscher laut einem Artikel auf Heise.de [Hei] nicht nur als Szene aus einem James Bond Film[1] denkbar. Da ein GPS-Signalfälscher als solches, nicht ohne großen Aufwand, realistisch in die Simulation eingebunden werden kann, wird dieser Fehler in einer abstrahierten Form implementiert. Dieser Sachverhalt bildet zudem eine gute Grundlage für ein Testszenario, in dessen Verlauf nach einer gewissen Vorlaufzeit ein Sprung in den Positionsangaben eingepflanzt wird. Anhand einer Plausibilitätsprüfung innerhalb der Software (des Virtuellen Prototyps) sollte ein solcher Sprung identifiziert werden können um ggf. entsprechende Gegenmaßnahmen einzuleiten.

Der anhand dieses Testszenarios abgebildete Fehler, spiegelt somit nur indirekt einen Spoofing-Angriff wieder. Der Hauptgedanke dieses Szenarios stellt ein Konzept dar, um einen Spoofing-Angriff durch eine externe Einpflanzung von fehlerhaften Positionsangaben zu erkennen. Da im ersten Szenario die Fehler trotz der externen Einwirkung durch α-Teilchen innerhalb der Hardwareschaltung entstanden sind, ist es notwendig, auch ein Szenario zu betrachten das einen inversen Ansatz verfolgt.

Um dies zu realisieren, wird im Rahmen des abstrahierten Szenarios ein fiktiver extern erzeugter Sprung als Fehler von außen dem System zur Verarbeitung zugeführt. Während der Verarbeitung soll der virtuelle Prototyp nun durch Abgleich der letzten drei Positionsangaben mit der aktuellen Position, mittels Plausibilitätsprüfung feststellen, ob fehlerhafte Positionsangaben eingepflanzt wurden oder nicht. Unter Verwendung der Fault-Injection kann so die Effektivität der implementierten Plausibilitätsprüfung überprüft und gegebenenfalls verbessert werden.

[1] Der Morgen stierbt nie - Im Laufe des Films, wird ein britisches Kriegsschiff durch gefälschte GPS-Signale in chinesische Hoheitsgewässer gelockt und dort versenkt

3.2.3 Implementierung der aus den Testszenarien abgeleiteteten Fehlertypen

Anhand der beiden Testszenarien fand bereits eine theoretische Ableitung und Abstraktion von Fehlertypen statt. In diesem Abschnitt erfolgt als kumulierte Fortsetzung dieser Diskussion eine Erörterung der Implementierung der drei abgeleiteten Fehlertypen im Rahmen des Experiments:

1. Beeinträchtigung der seriellen RS422-Übertragung durch α-Teilchen.
2. Entstehung eines Haftfehlers.
3. Auftreten eines Sprungs in der Positionsangabe durch einen GPS-Spoofing Angriff.

Zur Erzeugung des ersten Fehlers kommt eine Störfunktion zum Einsatz. Eine Störfunktion dient primär zur Abbildung von Fehlern in Software in Form einer eigens dazu geschriebenen Methode. Um nun diesen Fehler zu erzeugen, muss lediglich die Störfunktionsmethode an der entsprechenden Softwarestelle aufgerufen werden. Die Ausführung der hierzu implementierten Störfunktion erfolgt zufallsgesteuert. Sie ersetzt an einer zufälligen Position des NMEA-Datensatzes das gültige- durch ein beliebiges ASCII-Zeichen. Eine derartige Störung stellt ein typisches Indiz für eine fehlerhafte Übertragung dar und kann mittels Auswertung der Checksumme zielgerichtet aufgespürt werden.

Ein Haftfehler (Punkt 2) äußert sich in der hier abstrahierten Variante, durch wiederholte Ausgabe von ein und demselben Datensatz. Um dies zu realisieren, wird ein Down-Counter-Latch implementiert, der einen Haftfehler für die Dauer von bis zu 20 Sekunden erzeugt. Um eventuelle Falschmeldungen zu vermeiden, wird im virtuellen Prototyp erst nach dreimaligem Auftreten eines identischen Datensatzes Alarm geschlagen. Ab der Erkennung des Haftfehlers werden in der Regel Gegenmaßnahmen wie beispielsweise ein Neustart des GPS-Empfängers, durchgeführt. Der Einsatz von Gegenmaßnahmen ist jedoch nicht Gegenstand dieser Arbeit, daher wird lediglich die Erzeugung des Fehlers und dessen Erkennung betrachtet.

Bei dem unter Punkt drei beschriebenen Fehler, handelt es sich um einen extern erzeugten Fehler zur Simulation eines GPS-Spoofing Angriffs. Dieser Implementierung dieses Fehlers erfolgt ebenfalls mittels Down-Counter-Latch. Hierbei wird innerhalb eines zufälligen Zeitraums von maximal 25 Sekunden ein Sprung durch Manipulation der Positionsangaben im GPS-Generator erzeugt. Zur Erkennung verwendet der virtuelle Prototyp eine Plausibilitätsprüfung in Form eines ”Vertrauensintervalls”. Für den produktiven Einsatz müsste dieses Vertrauensintervall genauer bestimmt werden. Dies könnte beispielsweise durch eine umfassende empirische Analyse erfolgen.

3.2.4 Planung des Fault-Injection Experiments

Aufgrund der Anforderungen der beiden vorgestellten Testszenarien bietet es sich an, das bereits vorhandene Kommunikationsmodul als Medium für die Fehlereinpflanzung zu verwenden. Das implementierte Kommunikationsmodul wird für die Durchführung des Fault-Injection Experiments, entsprechend der Semantik der Fault-Injection Environment umfunktioniert (siehe Abschnitt 2.1.3).

Das Kommunikationsmodul überträgt die Empfangenen NMEA-Datensätze symbolisch mittels Bytestream vom GPS-Generator als fiktiver GPS-Receiver zum GPS-Parser, der einen Bestandteil der FCU abbildet. Da an dieser Stelle die Daten geballt auflaufen, stellt dieser ”Flaschenhals” eine geeignete Stelle zur Korrumpierung der Daten durch die Fault-Injection dar.

An diesem Einspeisungspunkt können die einzupflanzenden Fehler freier definiert und auf unterschiedlichen Abstraktionsniveaus aufbereitet werden, um diese in den GPS-Parser einzupflanzen. Durch diese Erkenntnis lässt sich der GPS-Parser, als das eigentliche System-Under-Test (SUT, dt. das zu testende System) in diesem Fault-Injection Experiment definieren. Dies verdeutlicht der nachfolgende Abschnitt 3.2.5 durch die Beschreibung des Aufbaus der Fault-Injection Environment.

Neben der eben vorgestellten Variante Fehler einzupflanzen, existieren diverse weitere Optionen als Einspeisungspunkt. So kann durch die Verwendung von Fehlern die auf beliebigen Abstraktionsniveaus liegen, indirekt eine Aussage über die Robustheit anderer kollaborativer Komponenten getroffen werden. Hierbei erfolgt die Einpflanzung des Fehlers dann indirekt. Beispielsweise wäre denkbar, dass durch invariante Ausgabedaten des GPS-Parsers, die Robustheit des Flugdaten Loggers (FDL) und dessen Dateisystem überprüfbar ist.

3.2.5 Aufbau einer Fault-Injection Environment im Kontext des GPS-Moduls und den gewählten Testszenarien

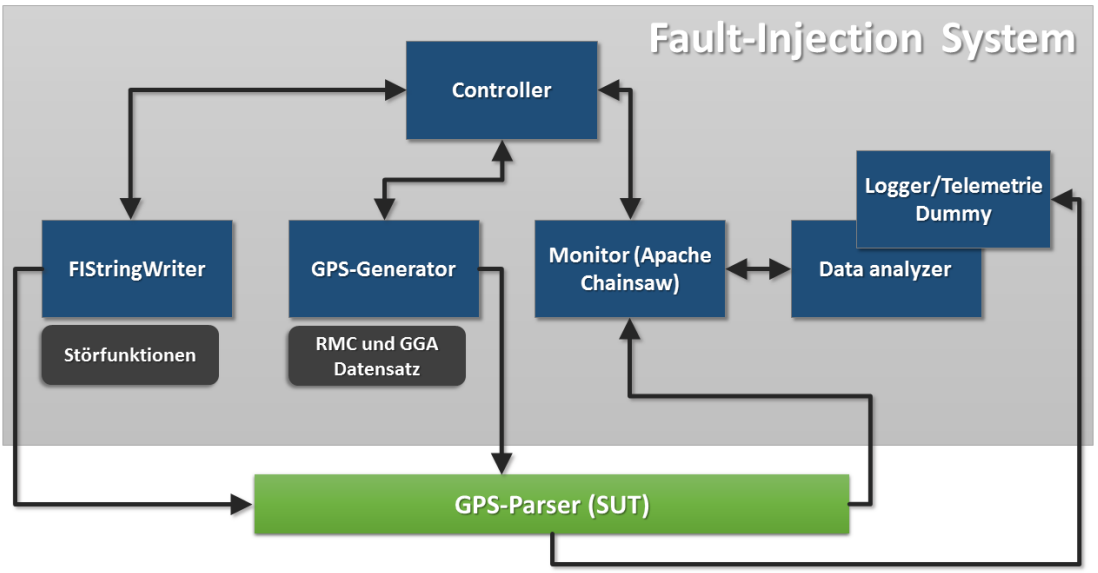


Abbildung 3.2: In Anlehnung an [HTI97, S.76]. Anpassung des Fault-Injection Environment an die gewählten Testszenarien und den virtuellen Prototypen

Die oben dargestellte Abbildung 3.2 wurde analog zu der im Kapitel 2.2 vorgestellten Fault-Injection Environment modelliert. Darüber hinaus spiegelt dieser Aufbau die Verwendung der einzelnen Komponenten des virtuellen Prototyps im Kontext der Fault-Injection Environment wieder. Hierbei wurden einige Komponenten, aufgrund ihrer (dualen) Semantik in das Gesamtbild der Fault-Injection Environment eingefügt. Darüber hinaus wird, wie bereits im Abschnitt 3.2.4 beschrieben, die GPS-Parser Komponente als SUT verwendet. Als workload generator dient der GPS-Generator, der weiterhin die NMEA-Datensätze RMC und GGA ausgibt. Die SLF4J und der Telemetrie Dummy werden als Data collector umfunktioniert, jedoch erfolgt die Analyse der Daten nicht automatisiert und muss folglich manuell durchgeführt werden.

Als Monitoring Tool wird der Logging Service Chainsaw von Apache als externe Java Anwendung eingesetzt, die ab JDK 1.5. als Ausführbare Datei im Development Kit enthalten ist. Da bereits die SLF4J respektive Log4J (ebenfalls ein Softwareproduk von Apache) verwendet wird, benötigt das Monitoring-Tool lediglich einen weiteren (socket) Appender.

Hierbei werden die Daten mit einem frei wählbaren Logging-Level (Info, Error, Debug) neben den bisherigen Logging Varianten (Konsole und Datei) zusätzlich via Stream vom virtuellen

Prototypen an das Monitoring-Tool Chainsaw transferiert und dort visualisiert. Um dies anschaulicher zu gestalten, dient die Abbildung 3.3 die einen Screenshot des Tools Chainsaw zeigt.

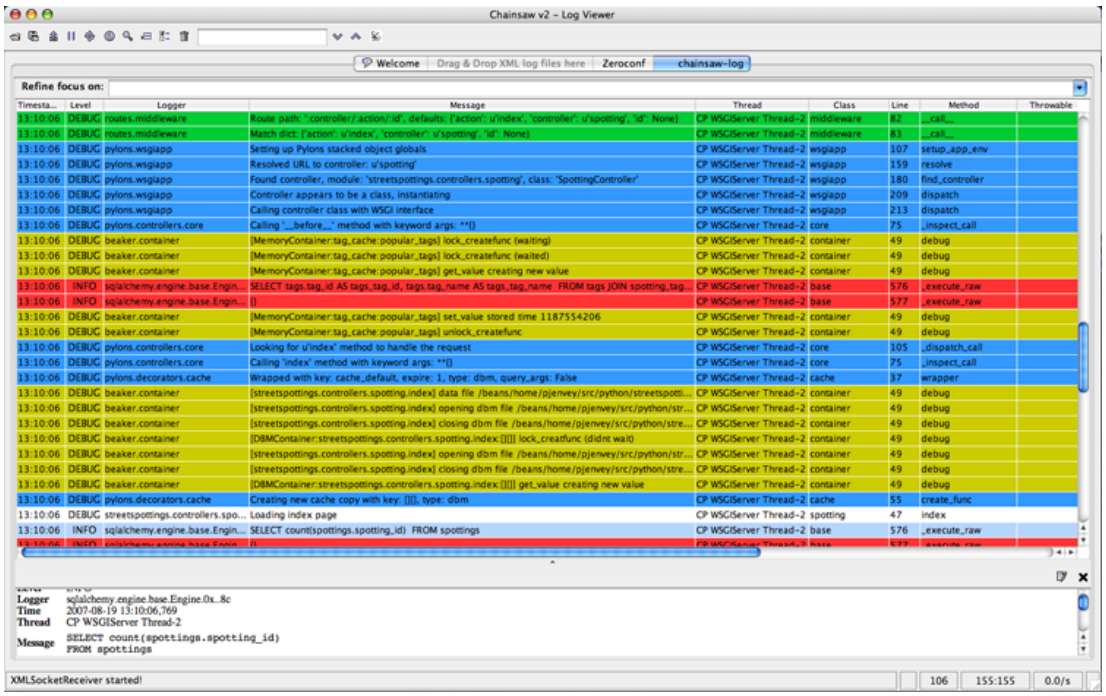


Abbildung 3.3: Exemplarische Darstellung des Monitoring-Tools Chainsaw von Apache [Cha]

Der Controller selbst ist statisch gestaltet, so muss in der Factory zur Erstellung des Fault-Injection Experiments der gewünschte Modus manuell ausgewählt werden. Hierbei kann zwischen den drei Optionen, die aus den beiden Testszenarios resultieren, gewählt werden. Es stehen als Option zur Verfügung:

1. Zufälliges Stören einzelner ASCII-Symbole der zu versendenden Daten, um eine Korrumpierung der NMEA-Datensätze durch α-Strahlung zu simulieren.

2. Simulation eines Haftfehlers, der durch α-Strahlung eine Verklemmung im GPS-Empfänger hervorgerufen hat.

3. Durchführung einer fiktiven GPS-Spoofing Attacke, durch einen ’‘Sprung” in den Positionsangaben der NMEA-Datensätze.

Die wohl wichtigste Funktionalität des Experiments stellt der Fault-Injector dar. Als Fault-Injector wird, wie bereits in Abschnitt 3.2.4 angesprochen, eine abgeänderte Version des Kommunikationsmoduls verwendet. Um eine versehentliche Vermischung beider Module zu vermeiden, erfolgt die Implementierung der beiden Module nach dem Singleton Entwurfsmuster. Das Singleton wird oft auch Einzelstück genannt und ist ein in der Softwareentwicklung eingesetztes Entwurfsmuster. Es stellt sicher, dass von einer bestimmten Klasse immer nur genau ein Objekt existiert [Fre+05]. Üblicherweise ist dieses Objekt global verfügbar. Ein solcher "Global State" erschwert die Qualitätssicherung mittels JUnit-Tests und sollte nur mit Bedacht verwendet werden. Als zusätzliche Absicherung vor einer versehentlichen, falschen Bedienung, befinden sich neben eindeutigen Namen beide Module auch in unterschiedlichen Packages. Darüber hinaus erfolgt die Instanziierung und Verwendung beider Komponenten in separaten Factories. Bei einer Factory handelt es sich ebenfalls um ein Entwurfsmuster. Es beschreibt, wie ein Objekt durch Aufruf einer Methode, anstelle eines direkten Aufrufs des Konstruktors erzeugt wird [Fre+05]. Damit ist sichergestellt, dass alle für die Instanziierung des Objekts notwendigen Aufrufe in der richtigen Reihenfolge erfolgen. Darüber hinaus werden so auch alle Kollaborateure der zu erstellenden Instanz sichtbarer.

Als Fehlerort in diesem Experiment ist der Bytestream vorgesehen. Hierbei erfolgt die Störung gezielt bei der Übertragung der NMEA-Datensätze vom GPS-Empfänger zum Parser (SUT). Um dies zu bewerkstelligen, wurden die Funktionalitäten für einen lesenden Zugriff auf die Daten im Kommunikationsmodul entfernt, so dass lediglich eine schreibende Funktion zur Verfügung steht.Darüber hinaus wurden die drei abstrahierten Fehlertypen zentral in die Funktionalität des FIStringWriters eingebettet. Anhand des virtuellen Prototyps lassen sich jedoch auch diverse andere Orte um Fehler zu induzieren finden, wie beispielsweise der selbst implementierte FileWriter zum Logging der durch den Parser analysierten Daten.

Anhand dieser zentralen Einbettung wertet die FIStringWriter-Komponente auch den gewählten Modus des Controllers aus und führt anschließend das gewählte Szenario durch. Da bewusst immer nur ein Fehlertyp pro Durchlauf zur Zeit existieren soll, unterstreicht dieser Sachverhalt die Notwendigkeit zur Implementierung der FIStringWriter-Komponente als Singleton.

3.3 Automatisierung von Mutationstests

Mutationstests werden in der Regel nach dem nachfolgenden Testablauf erstellt[VM98]:

1. Mutanten erzeugen.
2. Alle Testfälle auf das Originalprogramm anwenden.
3. Alle Testfälle auf die erzeugten Mutanten anwenden.
4. Äquivalente und getötete Mutanten (Testfälle die fehlschlagen) identifizieren und aussortieren.
5. Testfälle solange erweitern und auf die nicht getöteten Mutanten anwenden, bis alle Mutanten getötet werden.

Um den eben beschriebenen Ablauf greifbarer zu gestalten, wird in der nachfolgende Abbildung der eben beschriebene Testablauf graphisch dargestellt:

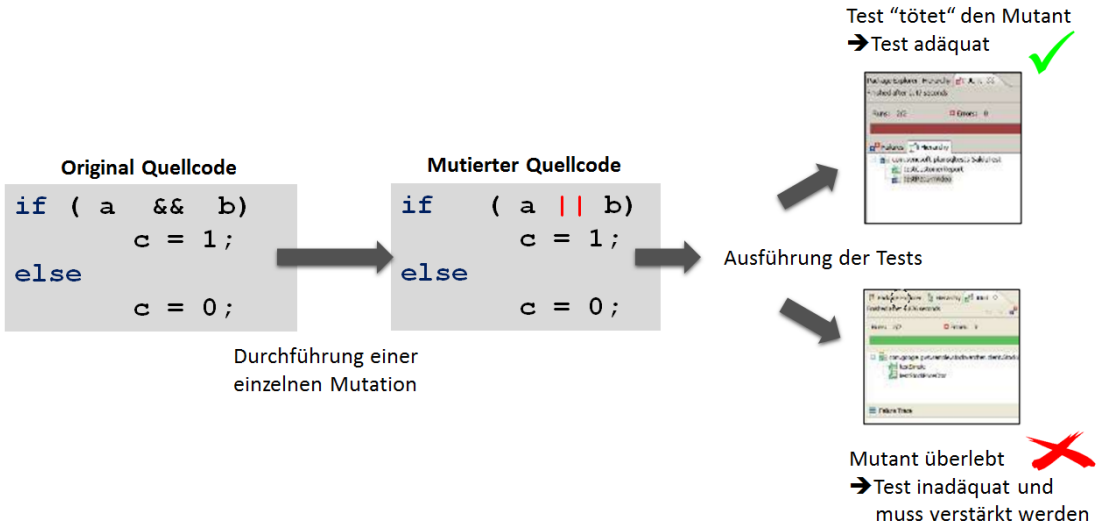


Abbildung 3.4: Graphische Darstellung des Mutationstestablaufs

Die Erzeugung der Mutanten beruht auf der Anwendung von diversen Mutationstransformationen. Da die Mutationstransformationen den zentralen Punkt im Mutationstest darstellen, wird in diesem Abschnitt zu einem späteren Zeitpunkt genauer Bezug darauf genommen. Anhand des Testablaufes ist bereits ersichtlich, dass der Mutationstest ein Instrument für die Bewertung der Leistungsfähigkeit von Testtechniken darstellt. Der Mutationstest verfolgt das Ziel, die

Qualität von Testfällen zu erhöhen. Dies geschieht, in dem sich der Mutationstest primär um auf syntaktische Fehler konzentriert, wie beispielsweise Tippfehler.

Ferner ist auch ersichtlich, dass beim Test von komplexen Systemen ein erhöhter Aufwand erkennbar ist. Dieser erhöhte Testaufwand lässt sich jedoch durch eine Automatisierung erheblich verringern. Vor diesem Hintergrund, befasst sich dieses Kapitel mit der Automatisierung von Mutationstests.

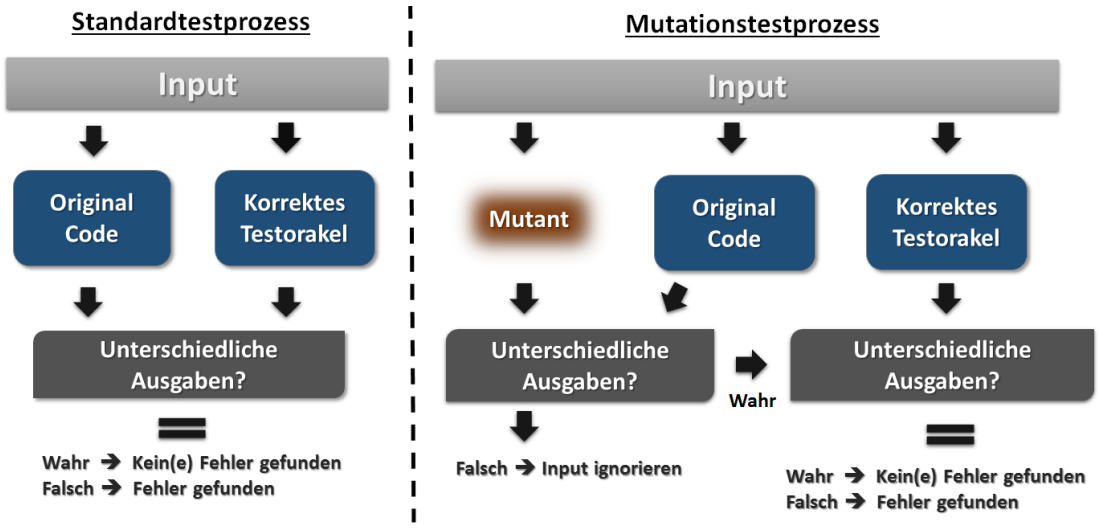


Abbildung 3.5: Darstellung in Anlehnung an [VM98, S.87]. Abgrenzung des Standardtestprozesses vom Mutationstestprozess

Anhand der Abbildung 3.5 kann man feststellen, dass im Mutationstestprozess lediglich der Unterprozess Mutant in den Standardtestprozess integriert wird. Bei einem Testorakel handelt es sich um ein Mittel zur Vorhersage eines Testergebnisses [Aic]. Mit Hilfe eines Testorakels und dem anschließenden Abgleich der Ausgabe aus der Software erfolgt dann die eigentliche Validierung der Software.

Das Prozedere eines Unit-Tests entspricht grob dem oben dargestellten Standardtestprozess. Eine xUnit-Architektur ermöglicht es verschiedene Elemente (Units) einer Software isoliert von anderen Programmteilen zu testen. Die Auflösung der getesteten Elemente kann von Funktionen und Methoden sowie Klassen, bis hin zu ganzen Komponenten reichen. Der wesentliche Vorteil einer solchen Architektur ist die Option einer breit gefächerten Automatisierung des Testprozedere. Es ist hierbei nicht nötigt, die Testergebnisse durch den Menschen zu ana-

lysieren, da die Assert-Methoden wie sie beispielsweise aus JUnit[2] bekannt sind, die dafür notwendigen Beurteilungen übernehmen.

Die Erzeugung von Mutanten im Zuge des Unterprozesses ”Mutant” der Abbildung 3.5 beruht auf den sogenannten Mutationstransformationen, die auf standardisierte Methoden wie beispielsweise der RSR-Transformation [VM98] basieren, bei der eine Mutation des Rückgabewertes einer Methode durchgeführt wird. Diese Transformationen lassen sich sehr gut in den bereits automatisierten Prozessablauf von xUnit-Architekturen integrieren. Die nachfolgende Tabelle gibt eine kleine Übersicht aus den typischen Mutationstransformationen:

AMC	Access mofifier change
AOR	arithmetic operator replacement
CRP	Constant replacement
DER	DO statement end replacement
IHD	Hiding variable deletion
IHI	Hiding variable insertion
IPC	Explicit call of a parent’s constructor deletion
LCR	Logical connector replacement
ROR	Relational operator replacement
SDL	Statement deletion
UOI	unary operator insertion
...	and many more

Tabelle 3.1: Auszug aus den in [VM98] vorgestellten Mutationstransformationen

Darüber hinaus bietet der Einsatz eines automatisierten Testtools, im Vergleich zu einer manuellen Durchführung, einen erheblichen Schutz vor Vermischung von Mutanten und Produktionscode. Um dies zu erreichen, setzen Tools wie beispielsweise μJava, Techniken wie beispielsweise die »Bytecode Translocation« [MOK05] ein. Die Bytecode Translocation untersucht und modifiziert den Bytecode, als intermediäre Darstellung des Java-Programms durch Verwendung einer Variation der Reflexion (engl. Reflection). In der Programmierung bedeutet Reflexion, dass ein Programm seine eigene Struktur kennt und diese, wenn nötig, modifizieren kann [Ull11]. Dadurch werden keine real existierenden Mutanten in Form von Java Code erzeugt. Diese Maßnahme schließt eine Vermischung oder Veränderung des originalen ”Produktionscode” dadurch aus, dass keine Daten auf die Festplatte geschrieben werden und der eigentliche Testprozess lediglich im Arbeitsspeicher stattfindet. Zur Veranschaulichung dient hier die Abbildung 3.6, dort wird die Implementierung der Bytecode Translocation des Mutationstestool μJava dargestellt.

[2] https://github.com/junit-team/junit/wiki/Assertions

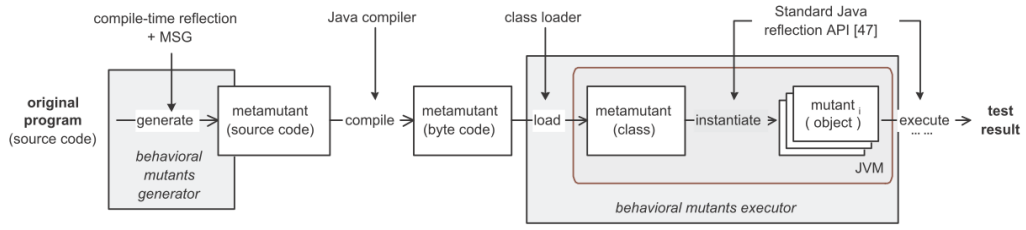


Abbildung 3.6: Darstellung übernommen aus [MOK05, S.9, Abb.2]. Darstellung der Implementierungsdetails des Mutationstest-Tools μJava

Anhand der Abbildung 3.6 kann man erkennen, wie die Bytecode Translocation abläuft. Während des Kompiliervorgangs wird unter Verwendung von Reflexion ein Metamutant anhand des originalen Quellcodes erzeugt und in die Java Virtual Machine (JVM) geladen. Die (automatisierte) Generierung des Mutationstest-Ergebnisses erfolgt durch Instanziierung von Mutanten-Objekten und deren Ausführung.

Darüber hinaus kann bei den meisten Tools wie μJava oder PIT-Mutationtest vorab eine Selektion der anzuwendenden Mutationstransformationen durchgeführt werden. Die Selektion verfolgt das Ziel, den Umfang der durchzuführenden Mutationstests zu definieren. Dies hat den Vorteil, dass durch die Automatisierung eine höhere Variation der Mutationen in kürzerer Zeit erzielt werden kann, als es manuell je möglich wäre. Durch eine höhere Variation der Mutanten können zudem die Testfälle detaillierter bewertet werden. Dies steigert letztendlich die Aussagekraft des durchgeführten Mutationstests und erhöht die Wahrscheinlichkeit für das Auffinden von schwachen Tests der Testsuite [MOK05]. Darüber hinaus bieten einige Tools eine Integration in Build-Management-Tools wie Ant, Apache Maven oder Gradle an. Dadurch entsteht beispielsweise im Zuge eines Nightly-Depolyments ein Fenster zur Durchführung von automatisierten Mutationstests.

Basierend auf der eben diskutierten Automatisierbarkeit und der immer wichtiger werdenden Integrationsmöglichkeit in Build-Management-Tools, beschränkt sich der praktische Teil der Mutationstest auf eine Evaluation von Tools. Als Basis zur Evaluation dient die vorhandene Testsuite des implementierten virtuellen Prototyps.

4 Modellierung eines Haftfehlers in VHDL

Haftfehler haben im Kontext der Hardwarebeschreibungssprachen (HDL) eine besonders wichtige Rolle. Sie treten insbesondere im Rahmen von Design- und Programmierfehlern auf. Da viele Lösungen in HDLs auf endliche Automaten (engl. Finite State Machine, kurz FSM) basieren, muss immer mit einer latenten Verklemmung gerechnet werden, die sich als Haftfehler bemerkbar macht. Daher befasst sich dieses Kapitel mit einem Exkurs um diverse Varianten zur Herbeiführung eines Haftfehlers zu Demonstrieren, anhand der Hardwarebschreibungssprache VHDL.

Die Entwicklung dieses Fehlermodells geht zurück in das Jahr 1959 und stellt das wohl am besten erprobte Fehlermodell dar [Eld59]. Mit dem Haftfehlermodell lässt sich eine Vielzahl von Defekten beschreiben, modellieren und simulieren. Die Modellierung selbst, folgt der Annahme, dass ein Eingang, Ausgang oder eine Leitung auf einem festen Wert liegt und kein Signalwechsel möglich ist. Ein großer Vorteil des Haftfehlermodells ist die Anwendbarkeit auf verschiedenen Abstraktionsebenen, die neben der Gatter-, auch die Registertransfer- oder Schalterebene umfasst. Formell lässt sich das Haftfehlermodell nach Fin und Fummi [FF00] wie folgt beschreiben:

$$\boldsymbol{sa}x, x \in \{0, 1\}$$

Unter dieser formellen Beschreibung versteht man das Abweichen eines Signals (sa) von seinem Sollwert. Im Sinne der Formel bedeutet dies, dass bei **sa**0 ein Stuck-At-0 Fault vorliegt und das betroffene Signal dauerhaft auf dem Wert '0' haftet. Analog dazu liegt bei **sa**1 ein Stuck-At-1 Fault vor, bei dem das Signal diesmal auf dem Wert '1' haften bleibt.

In der nachfolgenden Abbildung 4.1 ist eine Stuck-At Fault Modellierung exemplarisch anhand eines NAND-Gatters dargestellt. Die Kurzschlüsse K1 bis K3 mit dem Ground führen auf der Gatterebene zu Stuck-At-0 Faults. Der hier betrachtete Fall bildet nur Haftfehler ab, die am Ausgang beobachtbar sind. Für ein vollständiges Fehlermodell müssten neben dem Ausgang, die Eingänge A und B jeweils mit einem Haftfehler versehen werden. Da dies zu einer deutlich

zunehmenden Komplexität führt, wird in diesem Exkurs lediglich auf einen Haftfehler am Ausgang eingegangen. Die Wahl dieses Beispiels bietet unter anderem, im Zuge der Minimierung, die meisten Optionen an.

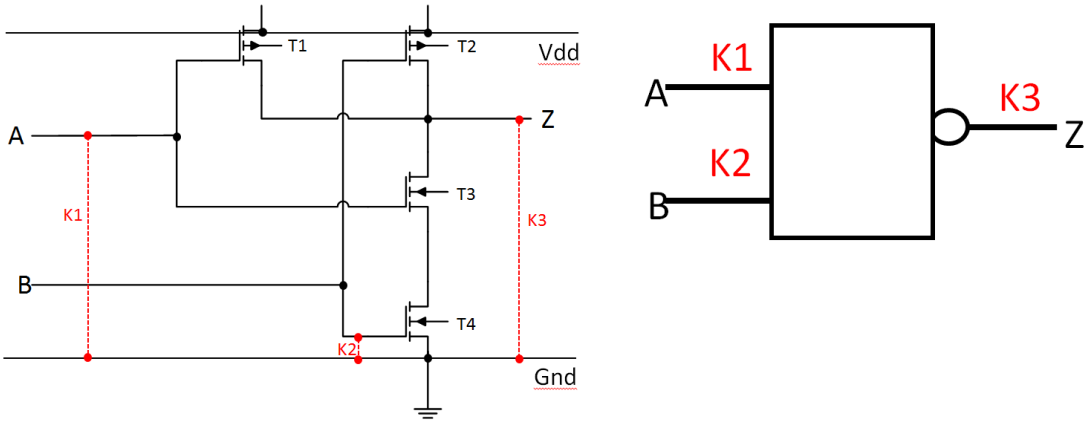


Abbildung 4.1: Beispiel für eine Haftfehlermodellierung die lediglich den Ausgang betrifft. Links: Schematische Darstellung auf Transistorebene mit den Kurzschlüssen K1 bis k3 Rechts: Darstellung der Problematik auf auf Gatterebene

Das in Abbildung 4.1 dargestellte Beispiel kann auf unterschiedlichen Varianten in VHDL modelliert werden. Die klassischste Variante ist, die Ableitung von VHDL-Code anhand einer Wahrheitstabelle. Um die Wahrheitstabelle zu erstellen, werden geeignete Eingangsbelegungen gewählt, um einen Fehler zu erkennen. Dies wird im Bereich VHDL meist als Testmuster oder auch Testvektor bezeichnet [RS09]. In der nachfolgenden Wahrheitstabelle ist das Resultat aufgelistet, das aus Kombination der Eingangssignale in Korrelation mit dem geplanten Fehlermodell aus Abbildung 4.1 erstellt werden kann:

A	**B**	**Z**	**Z(K1)**	**Z(K2)**	**Z(K3)**
0	0	1	1	1	0
1	0	1	1	1	0
0	1	1	1	1	0
1	1	0	1	1	0

Tabelle 4.1: Wahrheitstabelle für NAND und Haftfehler Z(K1) - Z(K3)

Für die generelle Durchführung einer vollständigen Simulation von Haftfehlern, muss nicht jeder Eingangs-Pin oder Schaltungsknoten im Zuge der Fault-Injection verwendet werden. Aufgrund diverser Analogien in den Permutationen der Eingangssignale, resultiert oftmals eine identische Semantik. Basierend auf diesen Analogien, können diese Kombinationen aus der Liste

der zu simulierenden Kombinationsoptionen gestrichen werden. Nachfolgend werden deshalb Optionen zur Minimierungen von Wahrheitstabellen näher definiert. Um die dahinterliegende Theorie pragmatisch zu untermauern, werden (sofern möglich) die Minimierung anhand des NAND-Beispiels demonstriert:

- Eine **Äquivalenz** besteht, wenn das Verhalten zwischen zwei oder mehreren Fehlern ein identisches Verhalten der Schaltung aufzeigt. Dadurch lassen sich diese äquivalenten Fehler zusammenfassen. Im NAND-Beispiel ergeben die beiden Fehler K1 und K2 ein identisches Ergebnis. Erkennbar ist dies anhand der äquivalenten Ergebnisse von Z(K1) und Z(K2) für den Ausgang Z.

- Ein **Dominanzverhalten** eines Fehlers gegenüber einem oder mehreren Fehlern impliziert eine Transitivität zwischen beiden. Wenn beispielsweise ein Fehler B Dominanz gegenüber einem Fehler A besitzt, dann lässt sich anhand eines Tests, der Fehler B aufdeckt auch Fehler A feststellen. Beim oben gewählten Beispiel besitzt der Fehler K3 eine Dominanz gegenüber gegenüber den beiden Fehlern K1 und K2.

- Als **Fehlerredundanz** wird ein Fehler bezeichnet, wenn er sich durch keine Eingangsbelegung nachweisen lässt. Solche Fehler entstehen innerhalb der vorliegenden Schaltung beispielsweise dadurch, dass nicht benötigte Eingänge auf feste Pegel gelegt werden. Daher ist es ratsam Fehler, die eine Fehlerredundanz aufzeigen, aus der Simulation auszuschließen. Jedoch ist die Feststellung eines solchen Fehlers schwieriger, als alle Fehlerfälle zu simulieren.

Anhand der vorhandenen Äquivalenz und Dominanz im NAND-Beispiel, kann die Anzahl der Fehler, die für die Haftfehlermodellierung infrage kommen, reduziert werden, etwa durch den einsatz eines Tools. Hier könnte beispielsweise der von Andreas Behling vorgestellte minimierende Fehlerlistengenerator [Beh00] weiterhelfen. Nach dem Aufstellen und minimieren der Wahrheitstabelle, kann anhand der minimierten Wahrheitstabelle ein VHDL-Code erstellt werden. So entstand der im Listing 4.1 Code aus der Wahrheitstabelle 4.1.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity NAND_GATE is
6  port(
7          A: IN bit;
8          B: IN bit;
9          Z: OUT bit
10 );
11 end NAND_GATE;
12
13 architecture BEHAVIOR of NAND_GATE is
14 -- Due to the equivalence of K1 and K2
15 -- both Signals can be combined
16 signal Z_K1_K2: boolean;
17 signal Z_K3: boolean;
18
19 begin
20         -- Process for modelling the stuck-at
21         -- fault (Based on the Lookup Table 4.1)
22         process (A,B) begin
23                 if Z_K1_K2 then
24                         Z <= '1';
25                 elsif Z_K3 then
26                         Z <= '0';
27                 else
28                         Z <= A NAND B after 10 ns;
29                 end if;
30         end process;
31
32 end BEHAVIOR;
```

Listing 4.1: VHDL Code basierend auf die Wahrheitstabelle 4.1

Nach Fertigstellung des Codes kann beispielsweise mit Hilfe des Simulators ModelSim eine Timingsimulation durchgeführt werden. Die nachfolgende Abbildung 4.2 zeigt exemplarisch ein Zeitverlaufsdiagramm, basierend auf dem in Listing 4.1 dargestellten VHDL-Code. Anhand der Abbildung ist laut der Wahrheitstabelle 4.1 erkennbar das bei der Kombination aus »A = '1'« und »B = '1'« der Ausgang auf einen Low-Pegel (0) wechselt. Die rot markierte Stelle

in Abbildung 4.2 zeigt, dass bei der Aktivierung des Fault-Injection Signals »ZK1_K2« der Ausgang auf einem High-Pegel (1) ”haften” bleibt.

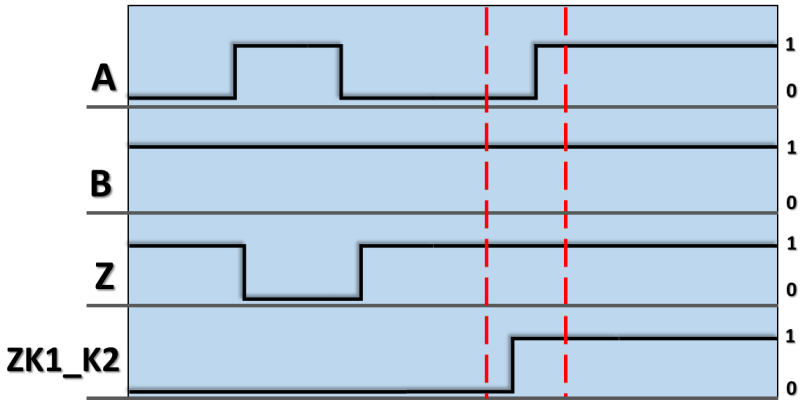


Abbildung 4.2: Exemplarisches Zeitverlaufsdiagramm zur Demonstration des Haftfehlers K1 bzw. K2

Neben der eben vorgestellten Variante gibt es auch diverse andere Möglichkeiten, einen Haftfehler in VHDL zu modellieren. Eine weitere Option ist die Verwendung des Assert-Befehls. Der nachfolgende Code im Listing 4.2 von Lothar Miller [Mil] zeigt einen exemplarischen Gebrauch dieses Befehls, mit dessen Hilfe für jedes Signal einer beliebigen Architektur ein Haftfehler erzeugt werden.

```
1    assert not(now>0 ns and z='1' and z'quiet(300 ns))
2        report "Output_Z_stuck_at_'1'_for_300ns"
3                severity warning;
4
5    assert not(now>0 ns and z='0' and z'quiet(400 ns))
6        report "Output_Z_stuck_at_'0'_for_400ns"
7                severity warning;
```

Listing 4.2: Exemplarischer Einsatz des Assert-Befehls nach Lothar Miller [Mil]

Abgesehen von dem Gebrauch bedingter Anweisungen oder dem assert-Befehl ist die Verwendung von logischen Operatoren zur Haftfehlermodellierung weit verbreitet. Diese Variante kommt beispielsweise dann zum Tragen, wenn ein Simulator die verwendete HDL-Beschreibung nicht interpretieren kann. Dadurch werden programmatische Lösungen, wie die bereits vorgestellten, hinfällig. Als Ausweg aus diesem Dilemma bietet sich der Einsatz von UND- und ODER-Logikgatter an. Die beiden nachfolgenden Darstellungen in Abbildung 4.3 zeigen anhand eines NAND-Gatters einen exemplarischen Einsatz beider Varianten.

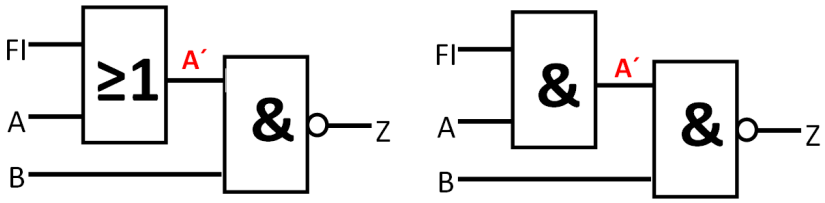


Abbildung 4.3: Schematische Darstellung der UND- bzw. ODER FI-Methode anhand eines NAND-Gatters

Um in den NAND-Gattern, welche in Abbildung 4.3 dargestellt sind, einen Haftfehler am Eingang A einzupflanzen, dient der Eingang FI am ODER- bzw. UND-Gatter. Durch die Aktivierung des FI-Eingangs mit einem High-Pagel, ensteht daraus ein Stuck-At-1 Fault am Eingang A‘. Analog dazu entsteht beim Einsatz eines UND-Gatters durch Anlegen eines Low-Pegels ’0’ ein Stuck-At-0 Fault.

Dieser Gedanke kann beliebig weitergeführt werden, so wäre es etwa auch denkbar, eine Schaltung mittels Clock-Logik zu betreiben, um diese auf das Verhalten von typischen VHDL-Programmierfehlern zu untersuchen. Nicht nur unerfahrene VHDL-Entwickler erzeugen unbeabsichtigt Clock-Logik aufgrund falscher Analogien zu anderen Programmiersprachen wie C oder Java. Das nachfolgende Listing 4.3 zeigt einen solchen Fehler bei der Erzeugung eines Daten Flip-Flops (DFF):

```
1 -- INVALID CODE for an asynchronous reset in VHDL
2 begin process(CLK,RESET)
3   if (CLK = '1' and CLK'event and ENABLE = '1'
4       and RESET = '0' ) then
5         -- Some assignments
6    else
7         -- Some other assignments
8    end if;
9 end process;
```

Listing 4.3: Häufig vorzufindender Programmierfehler der ein fehlerhaftes DFF mit Clock-Logik erzeugt

Das Problem in diesem Code liegt darin, dass der Code zwar kompilier- und synthetisierbar ist aber nicht der beabsichtigten Semantik entspricht. VHDL interpretiert jedes ”and” wörtlich. Daher wird das Clock-Signal mit dem ENABLE- und RESET-Signal durch UND-Gatter ver-

knüpft, bevor die Einspeisung des Clock-Signals im Clock-Eingang erfolgt. Die nachfolgende Abbildung 4.4 zeigt diesen fatalen Fehler detaillierter.

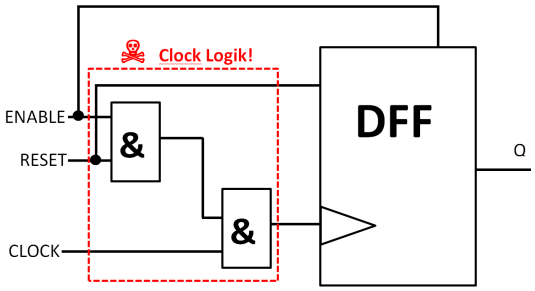


Abbildung 4.4: Schematische Darstellung des obigen

Wie man anhand der Abbildung 4.4 erkennen kann, nimmt sowohl das ENABLE-, als auch das RESET-Signal einen direkten Einfluss auf die Clock, der deutlich der Semantik eines Taktsignal (Clock) und einem DFF mit asynchronem Reset widerspricht. Durch die vermeintliche ”Vereinfachung” des Codes, wie man sie oft in anderen Programmiersprachen sieht, entsteht in VHDL unbeabsichtigt ein unerwartetes Verhalten. Dies stellt ein großes Problem dar, da ein solcher Fehler im Code als korrekt angesehen wird und sich dadurch beim Debugging als mögliche Fehlerursache entzieht. Die Schaltung zeigt erst nach dem Synthetisieren ein komplett unerwartetes Verhalten, welches sich in Form eines Haftfehlers äußert.

Eine weitere Variante ist in Abschnitt 7.5 des Anhangs vorgestellt im Kontext von Simulator Kommandos. Dort wird analog zu dem hier vorgstellten NAND-Beispiels anhand des ModelSim Built-In Kommandos »freeze« ein Haftfehler erzeugt.

Durch den Einsatz von Testbenches können die hier vorgestellten Fehler möglicherweise schneller gefunden werden. Allerdings stellen solche Komponententests kein Grundlage als Prüfkonzept für die Robustheit einer Schaltung dar. Gerade diese Tatsache macht es zunehmend wichtig, Methoden wie die eben präsentierten Haftfehlermodellierungen zu berücksichtigen. Diese Modellierungen bieten allerlei Raum für Ergänzung der vorhandenen Testszenarien. Daraus resultiert ein geeigneter Mechanismus um die Schaltung indirekt bei ihrer weiter Entwicklung robuster gegenüber den Auswirkungen von Haftfehler zu gestalten. Anhand dieses Sachverhalts, lässt sich als Fazit sagen, dass die Fault-Injection selbst nicht als direkte Testmethode anzusehen ist. Die Fault-Injection steht mit einem Komponententest in einer symbiotischen Beziehung. So dient diese Symbiose neben dem Aufdecken und Simulation von Fehlern auch zur nachweisbaren Immunisierung der Schaltung gegenüber bestimmten Fehlertypen und deren Auftreten.

5 Evaluation

In dieser Evaluation werden die in Kapitel 3 vorgestellten Testzenarien im Kontext des Fault-Injection Experiments bewertet. Ebenfalls spielt die Validierung der Funktionstüchtigkeit des virtuellen Prototypen eine wichtige Rolle. Neben der Auswertung des virtuellen Prototyps und des Fault-Injection Experiments findet in diesem Abschnitt auch die Evaluierung dreier Tools zur Durchführung von automatisierten Mutationstests statt.

5.1 Validierung des virtuellen Prototypen

Die Validierung der Funktionstüchtigkeit des virtuellen Prototypen erfolgt in zwei Schritten. Im ersten Schritt wird mittels exemplarischer Gegenüberstellung einer Log-Datei mit den generierten NMEA-Daten und der zugehörigen Log-Datei des Telemetrie Dummys die Korrektheit der durch den Parser durchgeführten, syntaktischen Analyse aufgezeigt. Hierbei spiegeln die NMEA-Daten im Klartext des Listings 5.1, die durch den GPS-Generator erzeugten Daten wieder. Auf Grundlage dieser Daten führt der implementierte Parser die syntaktische Analyse der Daten des Telemetrie Dummys durch. Im zweiten Schritt erfolgt der Einsatz eines eigens dazu geschriebenen Tools zur visuellen Darstellung von GGA-NMEA-Datensätzen. Mit Hilfe des Tools kann der simulierte ”Flug” graphisch dargestellt werden um einen besseren Bezug zu den kryptischen NMEA-Daten zu erhalten.

Das nachfolgende Listing zeigt einen exemplarischen Ausschnitt aus einer Log-Datei, die während des Evaluierungsprozesses entstandenen ist:

```
1  0     INFO  ParserThread  - $GPGGA,140530,5333.51,N,1001.48,E,8.0,02,2.0,14.9,M,0,M,,*7A
2  924   INFO  ParserThread  - $GPRMC,140531,A,5333.67,N,1001.6,W,010.1,392.0,140314,,S*54
3  924   INFO  ParserThread  - $GPGGA,140531,5333.67,N,1001.6,W,8.0,07,2.0,14.8,M,0,M,,*52
4  1937  INFO  ParserThread  - $GPGGA,140532,5333.82,N,1001.82,W,8.0,06,2.1,14.8,M,0,M,,*66
5  2753  INFO  ParserThread  - $GPRMC,140533,A,5333.86,N,1001.91,W,010.0,378.0,140314,,S*62
6  2963  INFO  ParserThread  - $GPGGA,140533,5333.86,N,1002.05,E,8.0,06,2.1,14.8,M,0,M,,*7D
7  3975  INFO  ParserThread  - $GPGGA,140534,5334.06,N,1002.17,W,8.0,01,2.1,14.8,M,0,M,,*63
8  4485  INFO  ParserThread  - $GPRMC,140535,A,5334.12,N,1002.27,E,010.4,19.0,140314,,S*46
9  4896  INFO  ParserThread  - $GPGGA,140535,5334.2,N,1002.32,E,8.0,08,2.1,14.7,M,0,M,,*45
10 5908  INFO  ParserThread  - $GPGGA,140536,5334.36,N,1002.5,W,8.0,03,2.1,14.6,M,0,M,,*5D
```

Listing 5.1: Auszug aus einer Log-Datei mit NMEA-Datensätzen

In Listing 5.1, sind die beiden (generierten) NMEA-Datensätze GGA und RMC gut ersichtlich. Die Zahlen der ersten Spalte spiegeln den zeitlichen Verlauf wieder. Anhand der verstrichenen Zeit in Millisekunden, zwischen der Generierung und Ausgabe der einzelnen Datensätze, lässt sich das Einhalten des Timings bei der Erstellung verifizieren. So erfolgt eine ungefähre Ausgabe eines GGA-Satzes innerhalb von 1Hz (Ausgabe des Datensatzes einmal pro Sekunde), sowie spätestens alle zwei Sekunden, die Ausgabe eines RMC-Satzes.

```
1  Latitude: 5333,51 | Longitude: 1001,48 | Time: 140530 | Quality: 8 | Direction: null | Altitude: 14,9 | Velocity:
       null
2  Latitude: 5333,67 | Longitude: 1001,60 | Time: 140531 | Quality: null | Direction: 392 | Altitude: null | Velocity
       : 10,1
3  Latitude: 5333,67 | Longitude: 1001,60 | Time: 140531 | Quality: 8 | Direction: null | Altitude: 14,8 | Velocity:
       null
4  Latitude: 5333,82 | Longitude: 1001,82 | Time: 140532 | Quality: 8 | Direction: null | Altitude: 14,8 | Velocity:
       null
5  Latitude: 5333,86 | Longitude: 1001,91 | Time: 140533 | Quality: null | Direction: 378 | Altitude: null | Velocity
       : 10,0
6  Latitude: 5333,86 | Longitude: 1002,05 | Time: 140533 | Quality: 8 | Direction: null | Altitude: 14,8 | Velocity:
       null
7  Latitude: 5334,06 | Longitude: 1002,17 | Time: 140534 | Quality: 8 | Direction: null | Altitude: 14,8 | Velocity:
       null
8  Latitude: 5334,12 | Longitude: 1002,27 | Time: 140535 | Quality: null | Direction: 19 | Altitude: null | Velocity:
        10,4
9  Latitude: 5334,20 | Longitude: 1002,32 | Time: 140535 | Quality: 8 | Direction: null | Altitude: 14,7 | Velocity:
       null
10 Latitude: 5334,36 | Longitude: 1002,50 | Time: 140536 | Quality: 8 | Direction: null | Altitude: 14,6 | Velocity:
       null
```

Listing 5.2: Auszug aus der Telemetrie Dummy Log-Datei mit den durch den Parser analysierten NMEA-Datensätzen

Das oben dargestellte Listing 5.2 zeigt die Ausgabe, der durch den Parser analysierten NMEA-Datensätze. Aufgrund der Beschaffenheit der einzelnen NMEA-Datensatztypen (siehe hierzu Abschnitt 2.3) stehen nicht immer alle Informationen zur Verfügung. Um das Fehlen des entsprechenden Datums zu repräsentieren, wird der Wert “null“ verwendet. Diese Darstellung hilft zu vermeiden, dass der Wert Null doppelt dargestellt wird. Bei einem Abgleich von Listing 5.2 mit dem Listing 5.1 kann man die Korrektheit der durch den Parser analysierten Daten nachvollziehen.

Anhand der beiden Listings 5.1 und 5.2 sind zwar die Datensätze auf Korrektheit und Plausibilität überprüfbar, jedoch ist die räumliche Darstellung der NMEA-Sätze im Kopf nahezu unmöglich. Um dies dennoch greifbar zu gestalten, wurde ein separates Programm entworfen. Hierbei wird als Basis der Visualisierung auf den GGA-Datensatz zurückgegriffen. Jeder empfangene Datensatz wird anschließend als Punkt repräsentiert, wodurch sich die Bewegung der fiktiven Flugdrohne nachvollziehen lässt.

Das nachfolgende Bild 5.1 zeigt die Ausgabe des gezeichneten Bildes, der zur Evaluierung verwendeten Log-Datei:

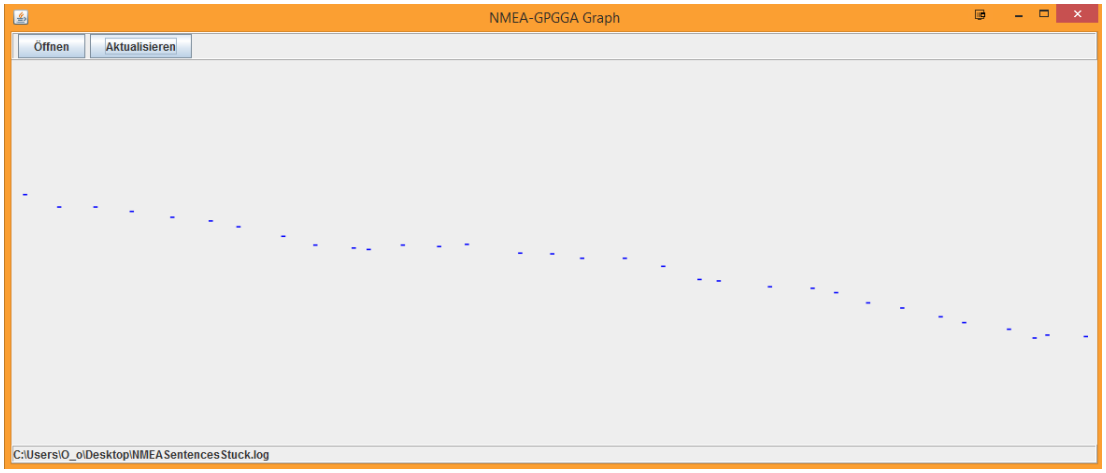


Abbildung 5.1: Visualisierung der NMEA-Logdatei mittels des NMEA-Graph Tools

Die Abbildung 5.1 stellt den fiktiven Flug graphisch dar. Die Unregelmäßigkeit der Abstände und Schwankung mit der absteigenden Tendenz der einzelnen Punkte beruht auf dem Modus “DESCENDING“ beim Randomisieren der Basis- bzw. Flugdaten.

5.2 Auswertung des Fault-Injection Experiments

Die Auswertung des Fault-Injection Experiments erfolgt als ”Proof of Concept” der in Kapitel 3.2.4 vorgestellten und im Rahmen des praktischen Teils dieser Arbeit implementierten Fault-Injection Environment. Um dies zu erreichen, werden die Testszenarios anhand von Screenshots verifiziert. Die Screenshots zeigen die Ausgaben des verwendeten Monitoring Tools ”Apache Chainsaw”, um die Funktionsweise des Konzepts in Korrelation mit den Testszenarien aufzuzeigen.

Als primäre Quelle zur Erkennung der eingepflanzten Fehler dient die in Kapitel 2.4 vorgestellte Checksumme. Wie bereits im Kapitel 3.2.1 beschrieben, dient die Auswertung der Checksumme als effektive Basis zur Erkennung von Invarianzen. Die hier präsentierten Resultate spiegeln das Ende einer längeren, evolutionär entstandenen Kette wieder. Hierbei wurden sukzessive Änderungen im Programmcode vorgenommen, die den virtuellen Prototyp gegen die eingepflanzten Fehler weitestgehend immunisierten.

5.2.1 Auswertung des α-Strahlen Experiments

Da anhand dieses Testszenarios zwei Fehlertypen abgeleitet werden können, erfolgt die Auswertung in zwei Schritten. Im ersten Schritt wird die Beeinträchtigung der RS422-Übertragung durch α-Teilchen diskutiert und im Anschluss daran das Haftfehler-Experiment.

Im Zuge der Fehlereinpflanzung und der daraus resultierenden Störung einer simulierten RS422-Übertragung wird auf eine eigens dafür geschriebene Störfunktion zurückgegriffen. Diese Störfunktion wurde bereits im Rahmen der Anforderungsanalyse beschrieben. Unter Verwendung dieser Störfunktion entstanden die in der nachfolgenden Abbildung dargestellten Fehler:

RELATIV...	TIMESTAMP	MARKER	LEVEL	LOGGER	MESSAGE	CLASS
3870	10:54:53		INFO	Unknown	$GPGGA,195909,5334.04,S,1001.97,W,8.0,02,2.1,14.8,M,0,M,,*7E	ParserFactory$Pars
4884	10:54:53		ERROR	Unknown	NMEA-Sentence malformed ($&PGGA,195910,5334.16,S,1008.19,E,8.0,04,2.0,14.7,M,¥,M,,*6A)	NMEAParser
4885	10:54:53	Malformed	INFO	Unknown	$&PGGA,195910,5334.16,S,1008.19,E,8.0,04,2.0,14.7,M,¥,M,,*6A	ParserFactory$Pars
4989	10:54:53		ERROR	Unknown	NMEA-Sentence malformed (^GPR(C,195910,A,5334.16,S,1 02.19,E,010.0,140.0,120314é,SV6A)	NMEAParser
4990	10:54:53	Malformed	INFO	Unknown	^GPR(C,195910,A,5334.16,S,1 02.19,E,010.0,140.0,120314é,SV6A	ParserFactory$Pars
5903	10:54:53		INFO	Unknown	$GPGGA,195911,5334.23,S,1002.4,E,8.0,06,2.0,14.5,M,0,M,,*51	ParserFactory$Pars
6916	10:54:53		ERROR	Unknown	NMEA-Sentence malformed ($6PGGA,195912,5334.3,S,100D.48,E,8.0,10,2.8,14.2,M,0,M,,*68)	NMEAParser
6916	10:54:53	Malformed	INFO	Unknown	$6PGGA,195912,5334.3,S,100D.48,E,8.0,10,2.8,14.2,M,0,M,,*68	ParserFactory$Pars
7029	10:54:53		INFO	Unknown	$GPRMC,195912,A,5334.3,S,1002.48,E,010.0,103.0,120314,,S*5F	ParserFactory$Pars
7941	10:54:53		INFO	Unknown	$GPGGA,195913,5334.46,S,1002.71,W,8.0,03,2.1,13.8,M,0,M,,*7E	ParserFactory$Pars
8959	10:54:53		INFO	Unknown	$GPGGA,195914,5334.53,S,1002.91,W,8.0,06,2.1,13.7,M,0,M,,*79	ParserFactory$Pars
9068	10:54:53		INFO	Unknown	$GPRMC,195914,A,5334.53,S,1002.91,W,010.0,257.0,120314,,S*78	ParserFactory$Pars
9875	10:54:53		INFO	Unknown	$GPGGA,195915,5334.73,S,1003.13,W,8.0,01,2.0,14.0,M,0,M,,*77	ParserFactory$Pars
10884	10:54:53		ERROR	Unknown	NMEA-Sentence malformed ($ÎPGGA,195916,5334.81,Ü,1003.27,W,8.0,07 2.0,13.7,M,0,M,,*65)	NMEAParser
10884	10:54:53	Malformed	INFO	Unknown	$ÎPGGA,195916,5334.81,Ü,1003.27,W,8.0,07 2.0,13.7,M,0,M,,*65	ParserFactory$Pars
11093	10:54:53		INFO	Unknown	$GPRMC,195917,A,5334.81,N,1003.27,W,010.1,356.0,120314,,S*64	ParserFactory$Pars
11904	10:54:53		INFO	Unknown	$GPGGA,195917,5334.95,S,1003.46,E,8.0,10,2.2,13.5,M,0,M,,*6F	ParserFactory$Pars
12919	10:54:53		INFO	Unknown	$GPGGA,195918,5334.99,S,1003.62,W,8.0,02,2.2,13.5,M,0,M,,*7B	ParserFactory$Pars
13128	10:54:53		INFO	Unknown	$GPRMC,195919,A,5334.99,S,1003.62,W,010.0,262.0,120314,,S*78	ParserFactory$Pars
13940	10:54:53		INFO	Unknown	$GPGGA,195919,5335.08,N,1003.73,W,8.0,11,2.4,13.4,M,0,M,,*6B	ParserFactory$Pars
14951	10:54:53		INFO	Unknown	$GPGGA,195920,5335.34,N,1003.95,W,8.0,10,2.3,13.0,M,0,M,,*64	ParserFactory$Pars
15061	10:54:53		INFO	Unknown	$GPRMC,195921,A,5335.34,N,1003.95,W,010.0,327.0,120314,,S*60	ParserFactory$Pars
15873	10:54:53		INFO	Unknown	$GPGGA,195921,5335.5,N,1004.04,W,8.0,06,2.3,13.1,M,0,M,,*5E	ParserFactory$Pars
16885	10:54:53		INFO	Unknown	$GPGGA,195922,5335.62,S,1004.12,W,8.0,08,2.2,13.1,M,0,M,,*79	ParserFactory$Pars
17094	10:54:53		INFO	Unknown	$GPRMC,195923,A,5335.62,S,1004.12,W,010.0,203.0,120314,,S*73	ParserFactory$Pars
17903	10:54:53		ERROR	Unknown	NMEA-Sentence malformed ($–PGGA,195923,533?.74,S,1004.27,W,8.0,03,2.3,1ç.8,M,0,M,,*7B)	NMEAParser
17904	10:54:53	Malformed	INFO	Unknown	$–PGGA,195923,533?.74,S,1004.27,W,8.0,03,2.3,1ç.8,M,0,M,,*7B	ParserFactory$Pars

Abbildung 5.2: Auszug aus Chainsaw mit visueller Darstellung der aufgetretenen Fehler bei der Simulation einer fehlerhaften Übertragung

Wie die obige Darstellung 5.2 zeigt, ist die Auswertung der Checksumme trotz des relativ trivialen Verfahrens (XOR-Bildung) äußerst effektiv bei der Fehlererkennung. Bei einer Entfernung der ”Schutzmechanismen”, wie die Auswertung der Checksumme, treten gravierende Probleme beim Parsing (Syntaktische Analyse) der NMEA-Datensätze auf, weshalb die Aufrechterhaltung eines Betriebs des NMEA-Parsers nahezu unmöglich ist. Die Probleme die auf etliche Laufzeitfehler zurückzuführen sind, wie beispielsweise das Konvertieren der Daten (String zu Double oder Integer), reichen vom Auftreten diverser Exceptions bis hin zu Abstürzen der Software. Somit lässt sich festhalten, dass ein effektiver Betrieb ohne eine Auswertung der

Checksumme im Fehlerfall sehr erschwert wird und der geringe Aufwand für die Auswertung der Checksumme zu einem robusteren Softwaresystem führt.

Die nachfolgende Abbildung zeigt im Rahmen der Simulation das Auftreten eines (abstrahierten) Haftfehlers:

RELATIV...	MARKER	LEVEL	LOGGER	MESSAGE	CLASS
5907		INFO	Unknown	$GPGGA,203940,5333.44,N,1001.42,E,8.0,07,2.0,15.1,M,0,M,,*77	ParserFactory$ParserThread
6519		INFO	Unknown	$GPRMC,203946,A,5334.36,N,1002.18,E,010.1,77.0,120314,,S*4F	ParserFactory$ParserThread
6930		INFO	Unknown	$GPRMC,203946,A,5334.36,N,1002.18,E,010.1,77.0,120314,,S*4F	ParserFactory$ParserThread
7948		INFO	Unknown	$GPRMC,203946,A,5334.36,N,1002.18,E,010.1,77.0,120314,,S*4F	ParserFactory$ParserThread
8358		ERROR	Unknown	### Stuck-At Bug Detected ###	NMEAParser
8359	stuck	INFO	Unknown	$GPRMC,203946,A,5334.36,N,1002.18,E,010.1,77.0,120314,,S*4F	ParserFactory$ParserThread
8969		ERROR	Unknown	### Stuck-At Bug Detected ###	NMEAParser
8971	stuck	INFO	Unknown	$GPRMC,203946,A,5334.36,N,1002.18,E,010.1,77.0,120314,,S*4F	ParserFactory$ParserThread
9881		ERROR	Unknown	### Stuck-At Bug Detected ###	NMEAParser
9883	stuck	INFO	Unknown	$GPRMC,203946,A,5334.36,N,1002.18,E,010.1,77.0,120314,,S*4F	ParserFactory$ParserThread
10189		ERROR	Unknown	### Stuck-At Bug Detected ###	NMEAParser
10190	stuck	INFO	Unknown	$GPRMC,203946,A,5334.36,N,1002.18,E,010.1,77.0,120314,,S*4F	ParserFactory$ParserThread
10900		ERROR	Unknown	### Stuck-At Bug Detected ###	NMEAParser

Abbildung 5.3: Auszug aus Chainsaw mit visueller Darstellung eines simulierten Haftfehlers

Hierbei ist ersichtlich, dass die Erkennung des verklemmten RMC-Datensatzes gemäß den Anforderungen des Testszenarios, erst nach dem dritten Auftreten als Haftfehler detektiert wird. Die Anzahl kann auch reduziert werden, allerdings könnte es so schneller zu Fehlalarmen kommen. Das Speichern der letzten drei Sätze des jeweiligen NMEA-Datensatzes bietet, neben einer Plausibilitätsprüfung der Daten, einen hohen Schutz bei einem insgesamt geringen und ressourcenschonenden Aufwand.

5.2.2 Auswertung des GPS-Spoofing Experiments

Die nachfolgende Abbildung stellt einen simulierten GPS-Spooing Angriff dar:

RELATIV...	MARKER	LEVEL	LOGGER	MESSAGE	CLASS
0		INFO	Unknown	$GPGGA,204633,5333.45,S,1001.46,E,8.0,01,2.0,15.0,M,0,M,,*64	Factories.ParserFactory
921		INFO	Unknown	$GPRMC,204634,A,5333.52,S,1001.61,W,010.0,296.0,120314,,S*79	Factories.ParserFactory
921		INFO	Unknown	$GPGGA,204634,5333.52,S,1001.61,W,8.0,03,1.8,15.0,M,0,M,,*78	Factories.ParserFactory
1934		INFO	Unknown	$GPGGA,204635,5333.59,S,1001.72,E,8.0,06,1.7,15.0,M,0,M,,*6B	Factories.ParserFactory
2450		INFO	Unknown	$GPRMC,204636,A,5333.63,S,1001.75,W,010.0,285.0,120314,,S*7E	Factories.ParserFactory
2959		INFO	Unknown	$GPGGA,204636,5333.66,S,1001.85,W,8.0,08,1.8,15.1,M,0,M,,*7E	Factories.ParserFactory
3871		INFO	Unknown	$GPGGA,204637,5333.74,S,1002.03,W,8.0,09,2.0,15.1,M,0,M,,*78	Factories.ParserFactory
3980		INFO	Unknown	$GPRMC,204637,A,5333.74,S,1002.03,W,010.0,298.0,120314,,S*77	Factories.ParserFactory
4890	Dashed	INFO	Unknown	$GPGGA,204638,6333.85,S,2002.11,W,8.0,08,2.1,24.9,M,0,M,,*73	Factories.ParserFactory
4893		ERROR	Unknown	### Dash has been detected ###	NMEAParser.NMEAParser

Abbildung 5.4: Auszug aus Chainsaw mit visueller Darstellung eines simulierten GPS-Spoofing angriffs

Die oben dargestellte Abbildung 5.4 zeigt die Erkennung eines Sprungs in den GPS-Positionsdaten. Ein solcher Sprung lässt sich gut graphisch mit Hilfe des bereits vorgestellten NMEA-Graph Tools visualisieren.

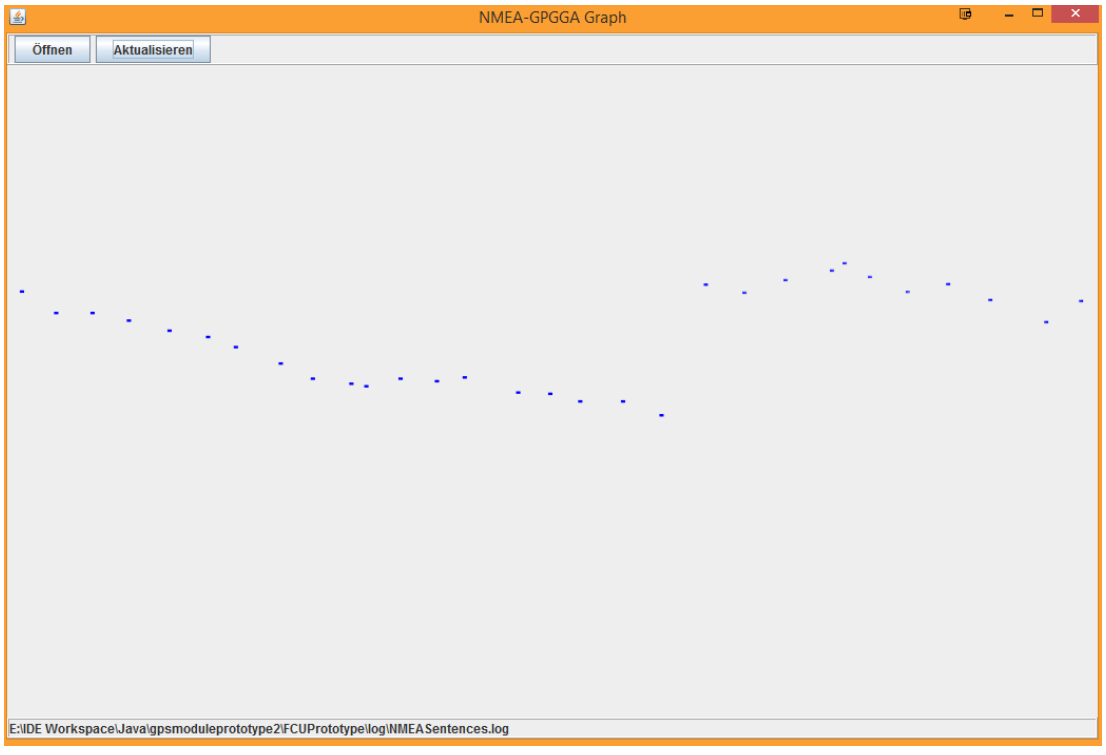


Abbildung 5.5: Visualisierung eines Sprungs mit Hilfe des NMEA-Graph Tools

Die Abbildung 5.5 zeigt einen großen Sprung in den NMEA-Positionsdaten. Hierbei kann man sich gut vorstellen, wie die Drohne während ihrem fiktiven Flug in die Irre geführt wird. Durch einen derartigen Angriff kann beispielsweise eine sich im autonomen Flugmodus befindliche Drohne gezielt gegen ein Hindernis gelenkt werden.

Ein solcher Angriff ist lediglich erkennbar, wenn eine größere Differenz in den Positionsangaben stattfindet. Das im virtuellen Prototypen verwendete Vertrauensintervall verwendet die letzte bekannte Position, um zu überprüfen, welche davon in einem realen Einsatz nur geringfügigen Schutz bieten würde - insbesondere, wenn im Zuge eines Angriffs kleinere Sprünge verwendet werden, die innerhalb des Vertrauensbereichs liegen. Anhand der Software, die im Rahmen dieser Arbeit entwickelt wurde, könnte ein geeigneteres Verfahren für die Erkennung eines GPS-Spoofing Angriffs gefunden und anschließend evaluiert werden.

Im Rahmen der Evaluierung des Proof of Concepts kann festgehalten werden, dass sich ein solcher Sprung in das System induzieren lässt. Allerdings besteht die eigentliche Schwierigkeit in einer ressourcenschonenden und effizienten Detektierung des Sprungs. Anhand dieses Testszenarios lassen sich weitere Testszenarien ableiten, wie beispielsweise eine Entführung der Flugdrohne.

Ein solches Entführungsszenario kann zwar simuliert werden, allerdings würde sich hier ein Angriff mittels gefälschten Differential-GPS-Daten im Zusammenspiel mit dem Original GPS-Empfänger anbieten. Um solche Daten zu generieren, müsste ein GPS-Jammer gebaut und betrieben werden. Dies wirft jedoch neben rechtlichen Fragen, auch Fragen bezüglich der Sicherheit von in der Nähe befindlichen Geräten und der lokalen Begrenzung des Störsignals (sofern dies durchführbar ist) bei der Testdurchführung auf.

5.3 Evaluation dreier Mutationstest-Werkzeuge für Java

Um Tests für Java Programme mittels Mutationstest-Tools auf schwache Tests zu überprüfen, existieren diverse Varianten. Um die Semantik eines schwachen Tests besser zu verstehen, dient das nachfolgende Listing:

```
1  /*Die zu testende Methode*/
2  public static String foo(int i) {
3    if ( i >= 0 )
4        return "foo";
5    else
6        return "bar";
7  }
8
9  /*Die Schwachstelle der nachfolgenden
10   JUnit-Tests ist die fehlende
11   Abdeckung  des Falls i == 0 */
12 @Test
13 public void shouldReturnBarWhenGiven1() {
14   assertEquals("bar", foo(1));
15 }
16
17 @Test
18 public void shouldReturnFooWhenGivenMinus1() {
19   assertEquals("foo", foo(-1));
20 }
```

Listing 5.3: Beispiel zur Darstellung eines schwachen Tests

Wie im Listing 5.3 beschrieben wird der Fall »i == 0« nicht abgedeckt. Solche schwachen Tests können durch den Einsatz von Mutationstest Tools automatisiert aufgedeckt werden. Als verfügbare Tools für Java lassen sich PIT Mutation Testing, Javalanche, μJava, Judy, Jumble und Jester benennen. Neben den eben benannten Tools stehen diverse weitere Tools zur Verfügung, die sich neben der Benennung der Operatoren für die Mutationstransformation auch in den angebotenen Features unterscheiden.

Als Basis für einen Vergleich der zu evaluierenden Tools dient μJava. Dieses Tool ist sowohl eines der ersten frei verfügbaren Tools wie auch Gegenstand vieler wissenschaftlicher Arbeiten und daher auch weitreichend erforscht. In diesem Zusammenhang lassen sich beispielsweise die beiden folgenden Arbeiten benennen: ”[Wil13] und [MOK05]” . Folglich kann angenommen

werden, dass neu entwickelte Tools effektivere Konzepte anhand der Resultate dieser Arbeiten implementieren konnten. Basierend auf dieser Annahme, dient die durch μJava empirisch ermittelte Erkennungsrate an schwachen Tests, als Richtwert für die Bewertung der Effektivität der Tools.

Ein weiteres Einschränkungskriterium stellt die Art und Weise dar, wie die Mutanten erzeugt werden. Hierbei kann zwischen der Quell- und Bytecode-Mutation unterschieden werden. Die Quellcode-Mutation findet durch das Einfügen der Mutation im original Quellcode statt. Anschließend erfolgt zur Validierung des zugehörigen JUnit-Tests, eine vom originalen Programmcode separierte Übersetzung und Ausführung des mutierten Codes. Die Funktionsweise der Bytecode-Mutation wurde bereits in Kapitel 3.3 im Zuge der »Bytecode Translocation « vorgestellt. Anhand der beiden Varianten, wurde PIT Mutation Testing als Vertreter der Bytecode-Mutation und Jester für die Quellcode-Mutation als vielversprechende Tools ausgewählt.

Die Grundlage für die Evaluierung selbst, bildet die JUnit-Testsuite, welche bei der Implementierung des virtuellen Prototyps erstellt wurde. Für eine bessere Überprüfbarkeit wurden zusätzlich zehn schwache Tests in die Testsuite eingefügt. Bevor die Ergebnisse der Evaluierung präsentiert werden, erfolgt zunächst eine Vorstellung der drei zu evaluierenden Tools »μJava, PIT Mutation Testing und Jester«.

5.3.1 μJava und μClipse-Plugin

Das Tool μJava stellt das Urgestein der Mutationstestsysteme für Java dar und wurde im Jahr 2003 von Jeff Offutt veröffentlicht. μJava ist das Ergebnis der Zusammenarbeit zwischen der Universität Korea Advanced Institute of Science and Technology (KAIST) und der George Mason Universität in den USA [Wil13].

Das Tool verwendet, wie bereits in Kapitel 3.3 beschrieben, den Bytecode als Ort für die Mutationserzeugung. Der Quellcode des Tools ist in einer limitierten Basis für Forschungen zugänglich. Die Entwicklung des Eclipse-Plugins μClipse wurde 2011 allerdings eingestellt. Jedoch steht der Quellcode auf SourceForge für eventuelle Weiterentwicklungen zur Verfügung. Das Plugin ist vom Entwicklungsstand weit genug vorangeschritten, um es produktiv im Rahmen kleinerer Projekte einzusetzen.

5.3.2 PIT Mutation Testing

PIT wurde ursprünglich als Framework für nebenläufige und verteilte Tests entworfen, bevor es in ein Mutationstesttool verwandelt wurde. Im Vergleich zu seiner Konkurrenz ist PIT ein relativ neues Tool. Es verfolgt laut den Autoren das Ziel, schnell und skalierbar zu sein und ist darüber hinaus als Ergänzung für den Ablauf in der Testgetriebenen Entwicklung (TDD) gedacht. Bei der TDD handelt es sich um eine agile Entwicklungsmethode, bei der die Entwickler konsequent vor dem schreiben des Codes, die zugehörigen Testfälle erstellten [Bec02]. Diese Tests werden mit PIT, als zusätzliche Instanz im Ablauf des TDD-Prozesses, auf Aussagekraft und Robustheit überprüft. Dadurch wird die Qualität der Tests und indirekt auch die Qualität der Software gesteigert [Col13].

Als Ort für die Mutation werden, ähnlich wie bei μJava, die Mutanten im Bytecode erstellt. Darüber hinaus wird eine inkrementelle Analyse als neues Feature zur Verfügung gestellt, bei der Änderungen in der Codebasis automatisch verfolgt werden. Darüber hinaus kann dadurch entscheiden werden, welche Methoden im nächsten künftigen Testlauf erneut mutiert werden müssen. Dadurch können die Ergebnisse von vorherigen Analysen und deren Konsequenzen wieder verwendet werden, um die Performance des Testvorgangs signifikant zu steigern.

5.3.3 Jester the JUnit test tester

Jester ist das erste open source Mutationstestsystem für Java. Das Tool führt die Mutationen auf der Quellcodeebene durch. Neben Jester existiert eine Variante des originalen Jesters namens Simple Jester. Das Tool Simple Jester stammt aus derselben Feder des Autors von Jester (Ivan Moore).

Simple Jester verfolgt, wie der Begriff ”simple” im Namen bereits verrät, das Ziel, die Benutzung des Tools zu vereinfachen. Diese Vereinfachung geht jedoch auf Kosten der Performance. Darüber hinaus wurden beide Tools seit längerem nicht mehr gewartet. Die letzte Aktualisierung erfolgte bei Jester im Jahr 2005 und bei Simple Jester im Jahr 2009 [Moo05].

Da die Mutationen bei Jester über eine sehr einfache Methode abgehandelt wird, in der Text gesucht und durch eine Mutation ersetzt wird. Bei dieser Methode kommt es zu diversen Komplikationen wie beispielsweise, dass der Mutant durch die Mutation mittels Jester nicht mehr kompilierfähig ist. Dadurch kommt es öfters zu einem Absturz von Jester, da Abhängigkeiten aufgrund mangelnder Kompilierbarkeit fehlen. Dies ist eine bekannte, aber ungelöste Problematik, die zu diversen Problemen während des Evaluierungsprozesses führten.

5.3.4 Evaluation der vorgestellten Tools

Evaluierungs Kategorie	μJava	PIT	Jester
Mutanten Erzeugung	Bytecode	Bytecode	Quellcode
Unterstützung für Java und Testframework	Nein	Ja	Nein
Eclipse-Plugin	(Ja)	Ja	Nein
Buildmanagement Tool Unterstützung	Nein	Ja	Nein
Lauffähigkeit der Mutanten	+	++	-
Vor dem Test induzierte schwache Tests	10	10	10
Insgesamt gefundene schwache Tests	13	21	9
Stabilität (Abstürze)	Nein	Nein	Ja
Aktive Weiterentwicklung	Nein	Ja	Nein

Tabelle 5.1: Darstellung der Evaluierungsergebnisse der untersuchten Mutationstest Tools

Anhand der Tabelle 5.1 wird ersichtlich, dass PIT Mutation Testing mit Abstand am besten abschneidet. Das schlechte Ergebnis von Jester kann neben der in die Jahre gekommenen Technik auch auf die Einstellung des Projekts zurück geführt werden. PIT verfügt neben diversen neuen Features auch über ausgereiftere Techniken. Teile der in PIT verwendeten Bytecode-Mutation weisen starke Analogien zu μJava auf. Das Tool μJava verfügt ebenfalls über viel Potential, das jedoch durch die Beendigung der Entwicklung nicht mehr weiter verbessert wird.

Der Bedarf an Buildmanagement Tools gewinnt in der modernen Softwareentwicklung immer stärker an Bedeutung. Gerade im Kontext der agilen Softwareentwicklungsmethoden, wie Continuous Delivery[1], werden Tools, die sich in den automatisierten Softwareverteilungsprozess (engl. deployment) integrieren immer essentieller. Dieser Umstand wird kurzfristig zu einer Verdrängung von Tools wie, μJava oder Jester führen, da aufgrund der fehlenden Entwicklungsaktivitäten mit einer Unterstützung von Buildmanagement Tools nicht mehr zu rechnen ist. Gerade im Zusammenspiel mit agilen Techniken, wie beispielsweise TDD und Continuous Delivery entfalten die Mutationstests ihr größtes Potential.

Als Fazit kann festgehalten werden, dass PIT Mutation Testing als ein vielversprechendes Tool angesehen werden kann. Gerade im Zusammenspiel mit agilen Softwareentwicklungsmethoden stellt es ein zukunftsträchtiges Tool zur Entwicklung für qualitativ hochwertige Software anhand einer Verstärkung der Testsuite dar. So profitierte die im Rahmen dieser Arbeit entstandene Testsuite stark anhand der durch die Tools aufgedeckten schwachen Tests.

[1] Continuous Delivery bezeichnet eine Sammlung von Techniken, Prozessen und Tools in der agilen Softwareentwicklung um den Prozess des Software Lifecycles zu verbessern. Darunter fallen auch Techniken wie die Testautomatisierung [DMG07]

6 Fazit und Ausblick

Dieses abschließende Kapitel fasst die gewonnen Erkenntnisse zusammen und erörtert sie. Darüber hinaus soll es auch einen Einblick in die möglichen Erweiterungen innerhalb dieser Arbeit entstandenen Software geben.

6.1 Fazit

In dieser Arbeit wurde als Pionierarbeit ein fiktiver Softwareprototyp entworfen, der als virtuelles Abbild eines kleinen Teils der durch die AES-Arbeitsgruppe geplante Hardwaresystemarchitektur anzusehen ist. Dieser virtuelle Prototyp versteht sich im Sinne der ”Virtual Hardware in the Loop” als Abbild von real existierender Hardware. Die Abbildung verfolgt das Ziel, eine Grundlage für einen effektiven Qualitätssicherungsprozess im Rahmen des AES-Projekts zu bilden. Hierzu wurden Teile des virtuellen Prototyps zu einer Fault-Injection Environment umfunktioniert, um damit ein Fault-Injection Experiment durchzuführen.

Um einen realen Bezug für die Fehlereinpflanzung zu erhalten, wurde das gesamte Projekt an zwei Testszenarien gebunden, anhand derer drei Fehlertypen abgeleitet werden konnten, die anschließend in den virtuellen Prototypen eingepflanzt wurden. Mithilfe der Fehlereinpflanzung war es möglich, frühzeitig ein Feedback, über die ”Verträglichkeit” der Fehler in Korrelation mit der Software zu gewinnen. Aufgrund der gewonnen Feedbacks resultierte eine größere und überprüfbare Robustheit des Testobjekts.

Neben der Fault-Injection, konnten in dieser Arbeit ebenfalls Gründe für die Automatisierbarkeit von Mutationstests analysiert werden. Anhand dieser Untersuchung erfolgte eine Evaluation dreier Werkzeuge, als deren Resultat sich PIT Mutation Testing als zukunftsweisendes Mutationstest Tool herauskristallisierte. Dieses Tool lässt sich im Rahmen einer möglichen Fortsetzung dieser Arbeit zur Qualitätssicherung der JUnit Softwaretests weiter verwenden. Mutationstests dienen nicht für die Fehlersuche im ausgeführten Code, sondern identifizieren Codestellen, die nicht oder unzureichend getestet werden. Daher eignet sich eine Integration des Tools, besonders in Korrelation mit agilen Softwareentwicklungsmethoden, wie Testdriven Development oder Continious Delivery.

Die im Rahmen dieser Arbeit durchgeführten Evaluierungen, zeigen neben einem proof of concept, dass der kreativen Entwicklung weiterer Testszenarien zur Fault-Injection keine Grenzen gesetzt sind. Anhand des erstellten virtuellen Prototyps und der zugehörige Fault-Injection Environment können weitere Konzepte erstellt werden, um Fragestellungen, die während des Entwicklungsprozesses der FCU-Systemarchitektur entstehen, noch vor Fertigstellung der Hardware zu lösen. Dadurch gewinnt die Systemarchitektur zunehmend an einer überprüfbaren Robustheit damit die Flugdrohne AC20.30 den, in ihrem Alltag drohenden Gefahren, sorgenfrei begegnen kann.

6.2 Ausblick

In den nachfolgenden Unterkapiteln werden mögliche Erweiterungen für diese Arbeit beschrieben. Während der Erstellung des praktischen Teils entstand beim Entwurf der Testsuite die Frage, ob Mocking auch als Fault-Injection Werkzeug verwendet werden kann. Diese Fragestellung wird im letzten Teil dieser Arbeit vorgestellt und diskutiert.

6.2.1 Weiterentwicklung des virtuellen Prototyps

Neben der Verarbeitung von GPS-Daten zur Geolokalisierung spielen diverse andere Komponenten und deren Daten im Kontext der FCU-Systemarchitektur (siehe Abbildung 1.2) eine wichtige Rolle. So könnte der virtuelle Prototyp, mit beispielsweise dem Auslesen eines fiktiven Akkustatus oder dem Beschleunigungssensor versehen werden. Eine Erweiterung des Prototyps bringt darüber hinaus den Vorteil, dass mit dem Testen der Hardware schon vor der Fertigstellung der realen Hardwareversion der FCU begonnen werden kann. Aber auch unabhängig von der Qualitätssicherung ist die Soft- und Hardwareentwicklung mit einem ausgereifteren Prototyps in der Lage, eine Lösung für verschiedene Fragestellung zu finden. Anhand dem daraus entstehenden Zuwachs an best practises, kann von dem Einsatz eines virtuellen Prototyps bereits vor der eigentlichen Implementierung profitiert werden.

6.2.2 Automatisierung der Fault-Injection Environment

Durch eine Erweiterung der Fault-Injection Environment mittels neuer Testszenarien oder der Erweiterung des Workload Generators, bedingt durch die Verfügbarkeit neuer Komponenten des virtuellen Prototypens, wird es notwendig, diese zu automatisieren. Eine manuelle Auswertung ist bei einer Vergrößerung des Umfangs zu umständlich und benötigt analog zum Funktionsumfang mehr Zeit zur Auswertung. Daher bietet sich der Einsatz eines generischen Matching-Verfahrens an, um die Fault-Injection zu automatisieren. Da dieser Schritt den Rahmen dieser Bachelorarbeit sprengen würde, blieb dieser Bereich weitestgehend unberührt. Eine Automatisierung des Data analyzers bietet daher eine gute Grundlage für weitere Studien im Fachgebiet der Automatisierung.

6.2.3 Mocking als Fault-Injection Werkzeug?

Im Zuge der Testfallerstellung einiger JUnit-Tests, kam nach und nach die Frage auf, ob Mocking-Frameworks wie PowerMock oder Mockito auch ein geeignetes Fault-Injection Werkzeug darstellen. Hierbei entstand die Idee, im Rahmen des Testprozesses ein Mock-Objekt zu verwenden, um Fehler an schwer zu testenden Stellen einzupflanzen. Darüber hinaus ließe sich auch evaluieren, ob auf diese Weise schwer zu provozierende Ausnahmen und Codestellen getestet werden könnten. Allerdings sollte dafür eine strikte Beachtung der Dependency Injection (DI) während der Entwicklung vorausgesetzt werden.

Bei der Dependency Injection, handelt es sich um ein Konzept der objektorientierten Programmierung, das die Abhängigkeiten eines Objekts zur Laufzeit reglementiert. Als echte Erleichterung im Bereich der DI, lassen sich beispielsweise die DI-Container des quelloffenen Frameworks Spring benennen (http://spring.io/). Der Einsatz eines DI-Containers, könnte einen geeigneten Ansatzpunkt zur Erarbeitung eines Konzepts im Rahmen dieser Fragestellung darstellen.

7 Anhang

In diesem Anhang werden neben einer kurzen geschichtlichen Zusammenfassung der Fault-Injection die drei Hauptvarianten der Fault-Injection beschrieben. Es existieren zudem noch zahlreiche weitere Hybridformen der Fault-Injection. Diese zu beschreiben würde jedoch den Rahmen dieser Arbeit resp. Anhangs sprengen.

7.1 Die Geschichte der Fault-Injection

Der erste Pionier in der Geschichte der Fault-Injection war Harlan D. Mills. Er veröffentlichte im Jahre 1972 einen technischen Report in dem er die grundlegende Technik für seinen Fault seeding Ansatz beschreibt [Mil72], die zuerst auf Hardware Niveau verwendet und eingeführt wurde. Diese Art von Fault-Injection wird auch als Hardware Implemented Fault Injection (HWIFI) bezeichnet und verfolgt das Ziel, Hardwarefehler innerhalb eines Systems zu simulieren. Später wurden diese Techniken auch auf Software übertragen und analog zur ursprünglichen Technik, als Software Implemented Fault Injection (SWIFI) bezeichnet [VM98].

Die Idee hinter dieser Technik ist es, die Zuverlässigkeit eines Programms zu bestimmen durch Schätzung der noch im Programm verbleibenden Fehler. Diese Schätzung wird von der Anzahl der eingepflanzten (seeded) Fehler abgeleitet, die während des Testens aufgedeckt werden sowie der kumulierten Anzahl der beim Testen tatsächlich gefundenen Fehler.

Hierzu formulierte Mills die nachfolgende Formel, um die geschätzte Anzahl der aktuell existierende Fehler im Programm zu bestimmen, unter der Prämisse, dass alle Fehler im Programm während des Testens nicht entfernt wurden:

$$q_k(N) = \frac{\binom{N_1}{r}\binom{N - N_1}{r - k}}{\binom{N}{r}}$$

Abbildung 7.1: Formel von Harlan D. Mills zur Bestimmung der geschätzten Anzahl an aktuell existierenden Fehlern in einem Programm [Mil72]

Bedeutung der einzelnen Variablen:
N_1 - Eingepflanzte Fehler in einem Programm
N - Anzahl der bereits vorhandenen Fehler die bereits bekannt sind
k - ist die Wahrscheinlichkeit das exakt k von r Fehler, während des Testens gefunden wurden und zu den eingepflanzten Varianten gehören

Die Maximale Wahrscheinlichkeit für die Schätzung der bereits vorhandenen (und bekannten) Fehler, gab er durch folgende Formel an:

$$N = \frac{N_1\big(r - k\big)}{k}$$

Abbildung 7.2: Formel zur Bestimmung der Maximalen Wahrscheinlichkeit der vorhandenen Fehler [VM98]

Anhand der beiden Formeln 7.1 und 7.2 kann die Beziehung zwischen den vorher eingepflanzten Fehlern, die während des Tests aufgedeckt wurden zu der Anzahl an generell gefundenen Fehlern dargestellt werden. Die Formel dient somit als Metrik zur Bewertung der Robustheit eines Systems. Anhand dieser Grundlagenforschung entstand als nächster großer Meilenstein die Software Mutation. Im Jahre 1980 wurde das erste Mutationstest Tool durch Timothy Budd im Rahmen seiner Doktorarbeit an der Yale University implementiert. Die Idee des Mutationstests geht zurück auf Richard Lipton [VM98].

Mit der Verfügbarkeit der heutigen Rechenleistung gewinnt das Konzept des Mutationstestens wieder an Bedeutung. Im Zusammenhang mit dem weiteren Voranschreiten der objektorientierten Programmiersprachen, endlichen Zustandsautomaten und dem Streben, eine Kontrollinstanz zu schaffen für die Überprüfung von Komponententests, machten auch den Einsatz von Mutationstests erforderlich. Daraus entstanden diverse neuere Konzepte, wie das Fuzzing (auch bekannt als Fuzz testing) die automatisiert bzw. teilautomatisiert ungültige, unerwartete oder zufällige Eingabedaten in ein Softwareprogramm zur weiteren Verarbeitung übergibt [Bud80]. Das Softwareprogramm selbst wird während der Prozedur des Fuzzing auf Abstürze, Speicherleckagen oder im Code verankerte assertions, die als Checkpoint

dienen, überwacht. Einer der ersten bekannten Einsätze von Fuzzing, war im Jahre 1983. In dem Artikel "Monkey Lives" [CA83] beschrieben die Macintosh Entwickler Steve Capps und Bill Atkinson, wie sie im Zuge der Entwicklung eines Journaling Features für eine Demotour des MacPaint Programms auf die Idee kamen, auf dessen Basis das Testtool "The Monkey" zu entwerfen. Steve Capps realisierte, dass die sogenannten journaling hooks, die im Rahmen einer automatisierten Demotour verwendet wurden, auch idealerweise für voraufgezeichnete Events in einem System verwendet werden können. The Monkey wurde daraufhin eingesetzt, um eine laufende Anwendung mit zufälligen Events zu füttern. Hierbei ist auch der Name für das Testtool entstanden, da die Anwendung sich während des Tests so verhielt, als ob ein wütender Affe die Maus und die Tastatur bedienen würde.

Dieser Ansatz wurde im Jahre 2010 durch Amazon Web Services (AWS) als bekannter Cloud Computing Provider wieder aufgegriffen und erhielt den Namen Chaos Monkey [Cia10]. Die Entwickler von AWS kämpften Monate lang mit dem Problem, dass einer der Server der Web Farm regelmäßig aufhörte auf externe Netzwerkanfragen zu reagieren. Trotz der frustrierenden Suche nach dem Bug, erkannten die Entwickler auch die positive Seite und entwarfen daraufhin die Testtoolsammlung Chaos Monkey. Die Testtoolsammlung folgt dem Motto: "Der beste Weg, um Fehler zu vermeiden ist, permanent zu scheitern". Basierend auf diesem Gedanken verfügt der Chaos Monkey über einen Scheduler der unterschiedliche Fehler zu vordefinierten Zeitpunkten ausführt. Die Fehler umfassen hierbei Auswirkungen wie beispielsweise das Beenden von Threads oder Netzwerkverbindungen. Diese Fehler werden während ihres Auftretens mit einem Monitoring Tool aufgezeichnet. Dadurch ist es möglich, sich gegen unerwartete Fehler, oder gar eine drohende Katastrophe, wie ein Totalausfall, zu wappnen und die vorhandenen Fehlerbehandlungsstrategien zu testen. Die Software ist inzwischen open source und wird neben den Netflix Ingenieuren durch eine breite Community weiterentwickelt (Stand 20.1.2014).

Im Jahre 2011 fand die Fault-Injection auch Erwähnung in der ISO Norm 26262. Diese ISO-Norm behandelt alle sicherheitsrelevanten Aspekte für elektronische Systeme in Kraftfahrzeugen. Darin ist ein Vorgehensmodell definiert, dass die diversen anzuwendenden Methoden für die funktionale Sicherheit eines Systems während der Entwicklung und Produktion sicherstellt.

7.2 Hardware Implemented Fault-Injection (HWIFI)

Die Hardware Fault-Injection (HWIFI) wird in der Literatur oft auch als Hardware-basierte Fault-Injection bezeichnet [HTI97]. Die Einpflanzung der Fehler findet bei dieser Technik auf physikalischer Ebene statt und wird in der Regel über die eigene Hardware injiziert. Diese eingepflanzten Fehler führen oftmals während der Fehlerverarbeitung durch das Hardwaresystem zu einer Veränderung von physikalischen oder elektrischen Eigenschaften. Dadurch kann eine Zerstörung des Zielsystems nicht ausgeschlossen werden.

Zur Durchführung der Fault-Injection gibt es verschiedene Varianten. Als einfachste Variante lässt sich die kontaktbehaftete Einpflanzung nennen. Hierbei wir der Fehler an den Pins der Chips induziert durch Anlegen eines bestimmten Pegels. Diese Störeinbringung wird daher auch als **pin-level Fault-Injection** bezeichnet. Überbrückungen werden allerdings auch bei den sogenannten **briding-faults** analog eingepflanzt. Auf diese Weise können in Bezug auf die Fehlerdauer sowohl dauerhafte, als auch periodisch wiederkehrende Fehler erzeugt werden. Die eigentliche Hürde besteht beim Einsatz von hochintegrierten Schaltkreisen darin, dass die Reaktionen nach dem Einpflanzen des Fehlers oft nicht ausreichend wiedergegeben werden können. So kann die Auswirkung am Ausgang zwar mit Hilfe eines Oszilloskops beobachtet werden, jedoch ist es nicht möglich, die genaue interne Reaktion zu beobachten.

Ein anderes, aber sehr häufig verwendetes Verfahren in der Hardware Fault-Injection ist die **strahlenbasierte Injektion**. Bei diesem Verfahren wird die Hardware mit energiereichen Partikeln bestrahlt. Diese Partikel verursachen sogenannte Single Event Upsets (SEUs), die sich als Bit-Flip-Fehler äußern. Die Idee hinter dem ersten Testszenario, bei dem die Auswirkungen von kosmischer Strahlung als simulierte Fehlerquelle Anwendung fand, bildet auch in dieser Fault-Injection Technik Strahlung den zentralen Kern. Bei der strahlenbasierten Fault-Injection wird die Hardware selbst durch ionische Strahlung, wie α-Strahlen oder Laserquellen [JMF98] bestrahlt und auf das Auftreten von SEUs untersucht.

Als letztes und gleichzeitig eines der wichtigsten Verfahren ist die **Scan Chain Implemented Fault Injection (SCIFI)**. In diesem Verfahren erfolgt die Fault-Injecton über sogenannte Scan chains (dt. Scankette oder oft auch als Scan Tests bezeichnet). Bei einer Scankette handelt es sich um ein Verfahren zur Überprüfung von digitalen Schaltungen auf strukturelle Defekte, Materialfehler und fertigungsbedingter Schäden. Um die Fehlerquellen etwas besser durchdringen zu können, dient die nachfolgende Abbildung 7.3, die diverse strukturelle Defekte und Materialfehler als eine Vergrößerung des Verbindungsnetzwerks einer Platine dargestellt. Etwa 90% der nachgewiesenen Fehler in integrierten Schaltungen stammen aus einem solchen Materialfehler oder einer beliebigen Kombination davon [Bal+96]

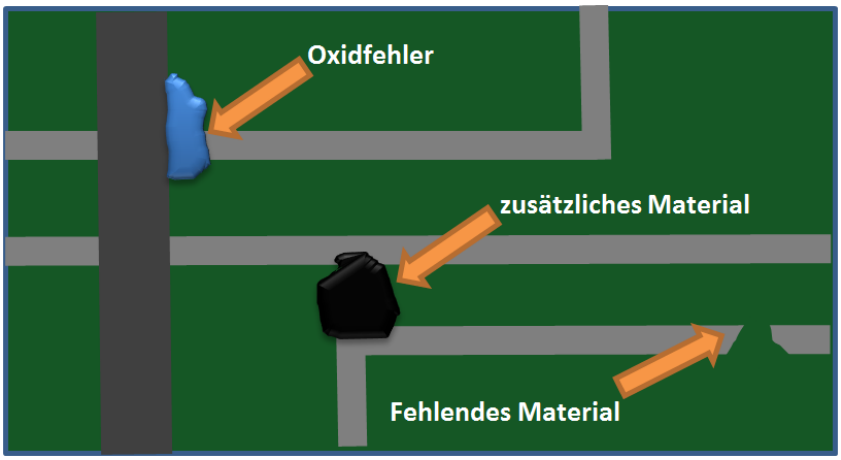


Abbildung 7.3: Darstellung verschiedener Materialfehler auf einer Platine bei starker Vergrößerung

Anhand der Abbildung 7.3 erkennt man, dass fehlendes oder zusätzliches Material, sowie fehlende Isolation durch Oxidfehler zu Kurzschlüssen oder Unterbrechungen führen. Gerade Oxidfehler sind auch eine häufige Ursache für Transistorfehler. Wenn das Gate-Oxid zu dünn ist, wird der Transistor anfälliger für eine Beschädigung durch Überspannung am Gate-Anschluss. Wenn hingegen zu viel Gate-Oxid vorhanden ist, wird das Schaltverhalten beeinflusst durch eine Verschiebung der Schwellspannung [Bal+96]. Solche Effekte werden neben der Hardware auch gezielt im Rahmen der Simulated Fault-Injection eingesetzt.

Um nun eine Prüfung auf Sturktur-, Fertigungs- und oder Materialschäden mit Hilfe einer Scankette durchzuführen, werden alle im Entwurf vorhandenen Flip-Flops mit einer Modus-Steuerung kombiniert. Mit dieser Modus-Steuerung kann zwischen einer hintereinandergeschalteten seriellen Betriebsart, dem sogenannten Shift-Betrieb und der regulären Betriebsart dynamisch gewechselt werden. Im Testvorgang kann ein beliebiges Testmuster in die digitale Schaltung injiziert werden, hierbei wird das erste und letzte Flip-Flop mit einer eigens dafür vorhergesehenen Ein- bzw. Ausgangsschaltung versehen. Bei der Umschaltung zwischen Shift- und regulärem Betrieb ergibt sich in Abhängigkeit des angelegten Eingangsbitmuster ein vordefiniertes Ausgangsbitmuster. Tritt nun anstelle des erwarteten Ausgangsbitmusters kein oder gar ein anderes Bitmuster auf, kann auf einen Defekt resp. Fehler geschlossen werden.

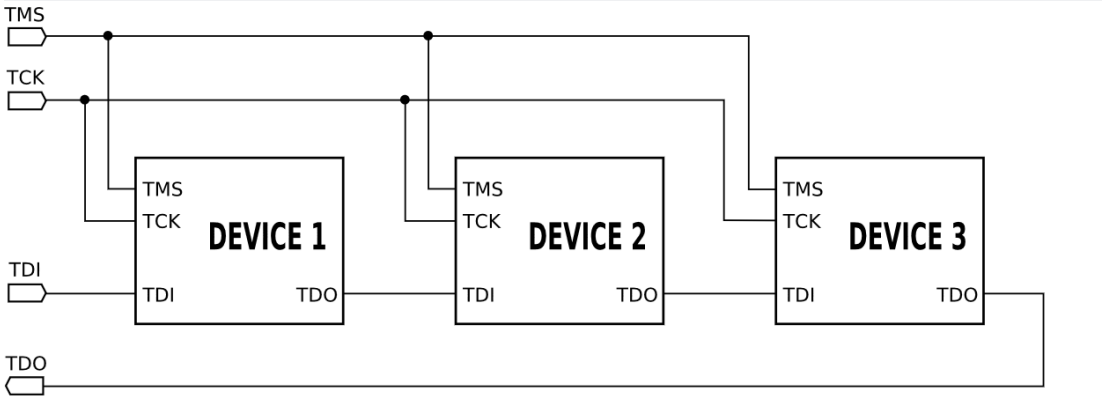


Abbildung 7.4: Darstellung einer Scan Chain mit drei Geräten. Übernommen von Grzegorz Pietrzak[Jta]

Die Abbildung 7.4 zeigt ein Scankette aus drei Testgeräten. Über die Ports TDI/TDO (Testdaten In/Out) werden die Testdaten in die Geräte induziert. Dies geschieht seriell, da alle Geräte hintereinandergeschaltet sind, wie in einer Kette, die die gesamte I/O-Struktur umfasst. Um die Verteilung der für den Betrieb notwendigen Steuerbefehle zu bewerkstelligen, dient der Port TMS. Durch diese Steuerbefehle lassen sich gezielt die gewünschten Test-Modi an den Geräten aktivieren. Bei jeder steigenden Flanke (TCK) werden jeweils die externen Daten (TDI) in die entsprechenden Register eingelesen.

Diese Art der HWFI kann relativ gut im Rahmen der Synthese und dem ATPG (Automatic Test Pattern Generation) mit dem Einsatz eines Tools automatisiert werden. Als Beispiel für ein Tool lässt sich hier die Xception Automated Fault-Injection Environment der Firma Critical Software Limited nennen [Xce]. Dieses Tool wird für diverse Bereiche, die über äußerst relevante sicherheitskritische Systeme verfügen verwendet wie beispielsweise Luft- und Raumfahrt, Logistik oder militärische Systeme.

7.3 Software Fault-Injection (SWFI)

Die Software Fault-Injection (SWFI) [HTI97] [VM98] wird wie die HWIFI in einem physikalischen System betrieben, wie beispielsweise einem Mikrocontroller, Desktop PCs oder einem virtuellen Prototypen wie er in dieser Arbeit verwendet wurde. Die Einpflanzung selbst erfolgt nicht in der Hardware, sondern innerhalb der Software die im Kontext dieses physikalischen Systems verwendet wird. Der große Vorteil dieser Technik liegt darin, dass keine zusätzliche Hardware benötigt wird. Die SWFI ist darüber hinaus flexibler als die HWFI und anwendbar ohne Zerstörung der Hardware. Eine Einpflanzung der Fehler erfolgt hierbei durch externe Induzierung der Fehler in Form von Code um die Verträglichkeit der Software auf diesen Fehler zu untersuchen.

Um die unterschiedlichen SWFI-Techniken zu generalisieren, können diese in die beiden Kategorien pre-runtime und runtime injection unterteilt werden. Die pre-runtime injection wird in der Literatur oft auch als compile-time injection bezeichnet, da die Fehler schon vor dem Kompilieren eingepflanzt werden. Auf diese Weise ist der Fehler statisch und kann sich nicht während der Laufzeit verändern. Basierend auf dieser Tatsache, eignet sich dieses Verfahren besonders zur Simulation von permanenten Fehlern. Da bei dieser Variante Haltepunkte sowie sämtliche Verzweigungen entfallen, besitzt diese Methode erhebliche Geschwindigkeitsvorteile bei der Ausführung.

Die runtime injection induziert ebenfalls die Fehler vor dem Laden des Codes in das Zielsystems. Allerdings können die Fehler während der Laufzeit durch die Verwendung von Haltepunkten oder Verzweigungen beliebig an- oder abgeschaltet werden. Dieses Verfahren hat im Vergleich zur pre-runtime injection den Nachteil, dass durch einen größeren Bedarf an Simulationszeit ein Overhead entsteht.

Die SWFI hat neben der vielen Vorzüge, wie beispielsweise die Unversehrtheit der Hardware den Nachteil, dass nicht immer alle Fehlerorte des Systems erreicht werden können. Während das auszuführende Programm sich meistens gut instrumentieren lässt, werden simulierte Hardwarefehler nur bedingt über die via Software zugänglichen Komponenten ermöglicht, wie beispielsweise Speicher oder CPU-Register. Aus Sicht der Software gestaltet sich die Fault-Injection in das Betriebssystem ebenfalls oft als schwierig. Einige Unix Betriebssysteme bieten hierfür die Möglichkeiten Fehler im Speicher durch Setzen von Flags zu simulieren. Jedoch sind Änderungen oder das Hinzufügen anderer Fehler aufgrund des limitierten Funktionsumfangs nicht ohne Vorkenntnisse des Kernels und der virtuellen Speicherverwaltung durchführbar.

Während der Entwurfsphase ist die SWFI anhand der Anforderungen und Systemspezifikationen sehr gut planbar. Ohne reale Hardware oder einem (virtuellen) Prototypen ist sie jedoch nicht einsetzbar. Dadurch ist die SWIFI im worst-case erst ab einem relativ späten Reifegrad des Zielsystems einsetzbar. Falls im Zuge des Testens eine Änderung von elementaren Funktionalitäten erfolgen müsste, könnten dadurch fatale und ökonomische Probleme entstehen.

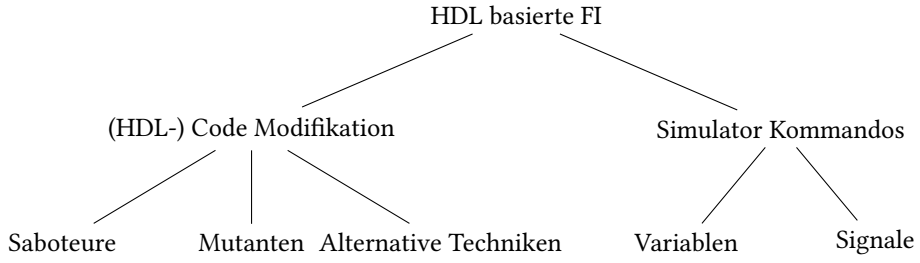
7.4 Simulated Fault-Injection nach ISO 26262

Unter Simulated Fault-Injection (SFI) wird aus der Hard- und Software des Zielsystems ein Modell geformt und das daraus resultierende Modell mit einem Simulator analysiert. Als Simulator selbst, können diverse Tools je nach Abstraktionsebene eingesetzt werden. Auf der elektrischen Abstraktionsebene kann ein Simulator, wie beispielsweise pSPICE [Psp] weiterhelfen, was jedoch eine programmatische Lösung ausschließt. Auf algorithmischer Ebene kann etwa Matlab in Kombination mit Simulink und diversen Plugins eingesetzt werden. Als Beispiel für ein populäres Plugin lässt sich an dieser Stelle das System Generator Plugin von Xilinx benennen. Mit Hilfe dieses Plugins lassen sich FPGA-Designs für Xilinx Hardware in Simulink durch Verwendung der angebotenen Bausteine im Drag & Drop Verfahren modellieren und simulieren. Anhand des modellierten Entwurfs kann nach Abschluss der Modellierung ein synthetisierbarer VHDL-Code automatisiert generiert werden.

Der konkrete Benefit dieser Methodik ist der Verzicht von realer Hardware. Die eigentliche Stärke der SFI liegt daher im Bereich der Hardwarebeschreibungssprachen wie VHDL oder Verilog. In diesem Bereich besticht die SFI-Methodik durch eine starke Steuerbarkeit, Erreichbarkeit und der guten Option zum Monitoring des Fault-Injection Vorgangs durch Simulationswerkzeuge, wie beispielsweise ModelSim. Zudem kann die SFI bereits während der Entwicklung auf unterschiedlichen Abstraktionsniveaus eingesetzt werden um Fehler einzupflanzen. Die SFI bietet die Option, sowohl permanente, als auch transiente Fehler in das zu entwickelnde Modell einzupflanzen. Ein großer Nachteil gegenüber der Hard- und Software Fault-Injection ist die enorme Menge an Zeit, die die SFI für die Durchführung und anschließende Auswertung der Simulation(en) benötigt.

7.5 Simulierte Fault-Injection (SFI) in Korrelation mit Hardwarebeschreibungssprachen

Basierend auf der guten Durchführbarkeit der simulierten Fault-Injection (SFI) in Korrelation mit Hardwarebeschreibungssprachen (HDLs), werden in diesem Kapitel einige Techniken und deren Konzepte diskutiert. Diese Konzepte könnten bei der Entwicklung und Qualitätssicherung der VHDL-Codes, die im Kontext der Softwarearchitekturentwicklung des AES-Projekts stehen signifikant unterstützen. Da diverse Abstraktionsebenen verwendet werden können, stellt die SFI ein geeignetes Werkzeug dar, um begleitend mit der Entwicklung, die Zuverlässigkeit und Robustheit des Modellentwurfs zu analysieren. Die nachfolgende Abbildung zeigt in Anlehnung an [Gra+01] eine Kategorisierung der unterschiedlichen Fault-Injection Varianten im Bereich der HDLs:



Die Methoden im Bereich der **Code Modifikation** erzeugen ein syntaktisch korrektes, aber in Bezug auf Semantik und Verhalten fehlerhaftes Modell. Bei **Saboteuren** handelt es sich um spezielle Komponenten, die in das originale Modell integriert werden. Im Bereich der HDLs ändern die Saboteur-Module die Werte oder das zeitliche Verhalten von System internen Signalen. Um dies gewährleisten zu können, verfügen die Saboteure neben einem Ein- und Ausgang, über einen speziellen Steuereingang um die Fault-Injection anzustoßen. Sobald der Saboteur aktiviert wird, beginnt dieser mit der Abweichung vom Soll-Verhalten durch Veränderung des logischen und oder zeitlichen Verhaltens einer Signalübertragung. Darüber hinaus lassen sich auch Bit-Flip-Fehler oder abgeleitet aus der Digitaltechnik der Wert ”Z”, der eine offene Leitung kennzeichnet, oder gar der Unbestimmt-Wert ”X” einpflanzen.

Die Saboteur Methode selbst wird auf zwei unterschiedlichen Varianten in das System integriert. Dabei unterscheidet man zwischen der seriellen und parallelen Methode. Bei der seriellen Methode wird die Saboteur Komponente in den Signalweg zwischen zwei Komponenten integriert, wohingegen bei der parallelen Methode die Saboteur Komponente zusätzlich an einen vorhandenen Signalweg angeschlossen wird. Beide Varianten werden nachfolgend diskutiert.

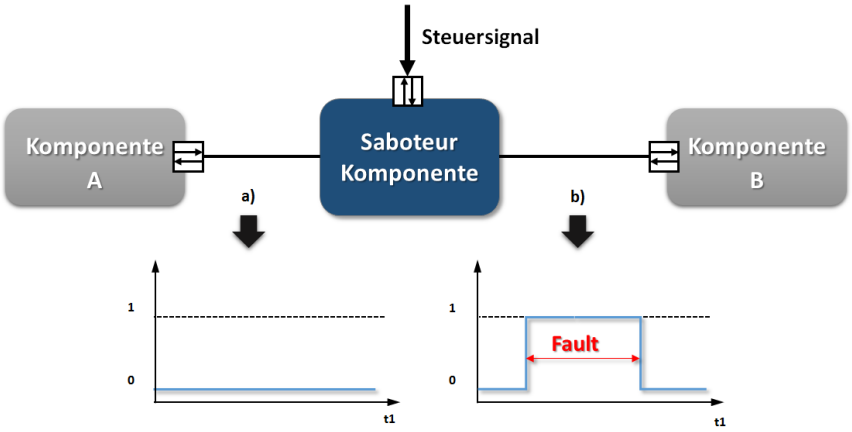


Abbildung 7.5: Schematische Darstellung der seriellen Saboteur Methode

Mit Hilfe der seriellen Methode, wie sie beispielsweise in der Abbildung 7.5 dargestellt ist, kann eine einfache Wertänderung erfolgen. In dem dargestellten Beispiel wandelt die Saboteur Komponente das in Punkt a) empfangene Rechenergebnis ”0” der Komponente A wie in Punkt b) beschrieben für eine Zeitlang in das Datum ”1” um. Die nun nachträglich modifizierte Berechnung von Komponente A wird nun an die Komponente B weitergereicht um dort verarbeitet zu werden. Hierbei kann der Empfang und die Verträglichkeit des Fehlers durch die Komponente B mittels Simulation analysiert werden. Es wären neben Wertänderungen auch andere Fehlertypen modellierbar, wie beispielsweise ein Haftfehler oder eine erhebliche zeitliche Verzögerung beim Transfer durch eine längere Zwischenspeicherung der durch die Saboteur Komponente empfangenen Daten.

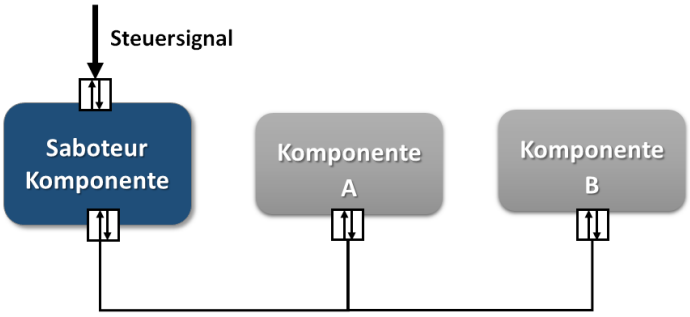


Abbildung 7.6: Schematische Darstellung der parallelen Saboteur Methode

Der Vorteil der parallelen Methode gegenüber der seriellen Methode liegt darin, dass mit dieser Variante auch Bussysteme gezielt gestört werden können. So wäre es denkbar ein Feldbussystem wie beispielsweise den Profibus damit zu testen, indem die Saboteur Komponente die Rolle eines defekten Sensor/Aktors übernimmt und auf diese Weise Störungen bei der Datenübertragung simuliert. Die Fault-Injection selbst erfolgt durch die Ausführung von Schreibvorgängen auf der gewünschten Leitung durch die Saboteur Komponente.

Der Einsatz von **Mutanten** führt analog zu den bereits vorgestellten Mutationstests (siehe Kapitel 2.1.4) zu einer Veränderung der Schaltung und stellt besonders in einem HDL Design eine Alternative zur Fault-Injection dar. Da Hardwarebeschreibungssprachen als ”Programmiersprachen” für das Schreiben einer Hardwarebeschreibung benutzt werden, entsteht beispielsweise beim Austausch von Logischen-Operatoren ein komplett neues Verhalten und Semantik der beschriebenen Hardware. Daher bietet es sich an, die bekannten textuellen Fehlermodelle aus dem Softwaretest auch auf die HDLs zu übertragen. Hierzu stellten die beiden Autoren Hayek und Robach [HR96] auf der Internationalen Testkonferenz in Washington DC diverse spezielle Mutations-Operatoren vor, die beispielsweise neben dem Ersetzen von Variablenzuweisungen auch arithmetische oder logische Operationen manipulieren können.

Unter dem Begriff **Alternative Techniken** werden alle Methoden zusammengefasst, die nicht in die dargestellte Klassifizierung der Abbildung 7.5 passen. Darunter fallen beispielsweise auch Fault-Injektion Tools, wie VERIFY das im Rahmen eines Reports [Bal+96] der Universität Erlangen vorgestellt wurde, oder Techniken wie die sogenannten Fault masks. Eine fault mask fällt auch unter die Signalkorrumpierung und repräsentiert die Bits die im Rahmen einer Fault-Operation angewendet werden. Wenn eine fault mask beispielsweise die Bits 1,4 und 7 abdeckt und die Fault-Operation ein XOR ist, dann wird ein Fehler durch Kippen des 1,4 und 7 Bits erzeugt. Dies kann im weiteren Sinne im Rahmen der Hardwaresimulation auch als eine Variation des Stuck-At Bugs angesehen werden.

Unter **Simulator Kommandos** versteht man die Verwendung von Built-In Kommandos, die in leistungsfähigen HDL-Simulatoren wie beispielsweise ModelSim Anwendung finden. Diese bieten eine Benutzer-Schnittstelle in Form einer Konsole an, die dem Anwender erlaubt mittels spezieller Kommandos den Simulationsvorgang interaktiv zu verändern. Es existieren dabei auch Befehle, die eine nahezu

willkürliche Veränderung von Signalwerten oder dem Inhalt von Variablen durchführen. Als Beispiel für die Modellierung eines stuck-at-1 Fehlers würde der Befehl, analog zum modellierten Stuck-At-1 Fehlers des nachfolgenden NAND-Beispiels (Kapitel 4) wie folgt aussehen:

$$forcefreezesim : /nand/A\_stuck\_at\_1 15$$

Erläuterung des obigen Kommandos: Das Kommando $forcefreeze$ friert das Signal $A\_stuck\_at\_1$ im Projekt sim, der Komponente $nand$ nach einer Verzögerung von 5ns auf den Wert 1 ein. Aber auch weitere Fehlersimulationen sind unter Verwendung von ModelSim denkbar, da der HDL-Simulator Skriptsprachen wie Tcl unterstützt. Aufgrund dieser Tatsache ist der Vorgang insgesamt auch automatisierbar auf Kosten von zusätzlicher Simulationszeit. Einer der wesentlichen Vorteile der Simulation Kommandos besteht darin, dass keinerlei Änderungen am Schaltungsdesign oder dem daraus resultierenden Code vorgenommen werden müssen. Die Änderungen können interaktiv an der Simulations-Wave vorgenommen werden, was sich insgesamt positiv auf die Simulationszeit auswirkt. Nachteilhaft ist jedoch zu benennen, dass nur Fehlermodelle abgebildet werden können, die mit der Semantik der Built-In Kommandos des Simulators übereinstimmen.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literatur

[AB80] Miron Abramovici und Melvin A. Breuer. *Fault Diagnosis on Effect-Cause Analysis: An Introduction.* 1980.

[Aic] Berndhard K. Aicherning. *Qualitätssicherung in der Softwareentwicklung - Automatisiertes Testen in Java.*

[Arl+90] J. Arlat u. a. *Fault Injection for Dependability Validation: A Methodology and some Applications.* Techn. Ber. IEEE Transactions on Software Engineering, 1990.

[Atw10] Jeff Atwood. *Working with the Chaos Monkey.* 2010. URL: http://blog.codinghorror.com/working-with-the-chaos-monkey/.

[Bal+96] F. Balbach u. a. *Simulationsbasierte Zuverlässigkeitsanalyse Internal Report No.: 6/96.* Techn. Ber. Universität Erlangen, 1996.

[Bec02] Kent Beck. *Test Driven Development by Example.* Addison Wesley Pub Co Inc, 2002.

[Bec08] Kent Beck. *Implementation Patterns.* Addison-Wesley, 2008.

[Beh00] Andreas Behling. "Entwicklung eines minimierenden Fehlerlistengenerators". Magisterarb. BTU Cottbus, 2000.

[BP03] Alfredo Benso und P. Prinetto. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation (Frontiers in Electronic Testing).* Springer US, 2003.

[Bud80] Timothy Alan Budd. *Mutation Analysis of Program Test Data.* Techn. Ber. 1980.

[Byt] *ByteMan (JBoss).* URL: http://www.jboss.org/byteman.

[CA83] Steve Capps und Bill Atkinson. *Monkey Lives.* 1983. URL: http://www.folklore.org/StoryView.py?project=Macintosh&story=Monkey_Lives.txt.

[Cha] *Chainsaw Tutorialseite.* URL: http://docs.pylonsproject.org/projects/pylons-webframework/en/latest/logging.html.

[Cia10] John Ciancutti. *5 Lessons Weve Learned Using AWS.* 2010. URL: http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html.

[Col13] Henry Coles. *PIT Mutation Testing*. 2013. URL: http://pitest.org/.

[Dem10] W. Demtröder. *Experimentalphysik 4: Kern-, Teilchen und Astrophysik*. Springer Verlag, 2010.

[DMG07] Paul M. Duvall, Steve Matyas und Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison Wesley Pub Co Inc, 2007.

[Eld59] R. D. Eldred. *Test Routines Based on Symbolic Logical Statements*. Journal of the ACM, vol. 5, no. 1, pp33-36, 1959.

[FF00] Allesandro Fin und Franco Fummi. *A VHDL Error Simulator for Functional Test Generation*. Techn. Ber. Universität von Verona, Tagungsband der Konferenz für Design, Automatisierung und Test in Europa, 2000.

[Fis01] Paul Fischer. *Grafik-Programmierung mit Java-Swing*. Addison-Wesley, 2001.

[Fre+05] Eric Freeman u. a. *Entwurfsmuster*. O'Reilly, 2005.

[Fre05] Stefan Freinatis. *Towards Comparability in Evaluating the Fault-Tolerance of Safety-Critical Embedded Software*. Techn. Ber. Universität Duisburg-Essen, 2005.

[Gar] *GPS 18x Technical Specifications (GARMIN)*.

[Gra+01] J. Gracia u. a. *Comparison and Application of Different VHDL-Based Fault Injection Techniques*. Techn. Ber. IEEE International Symposium on Defect und Fault Tolerance in VLSI Systems (DFT'01=, 2001.

[Hei] *Der GPS-Hack*. URL: http://www.heise.de/tr/artikel/Der-GPS-Hack-275686.html.

[HM04] G. Hermann und D. Müller. *ASIC - Entwurf und Test*. Chemnitz: Fachbuchverlag Leipzig, 2004.

[HR96] Ghassan Al Hayek und Chantal Robach. *From Specification Validation to Hardware Testing: A Unified Method*. Techn. Ber. Tagungsband: International Test Conference, 1996.

[HTI97] Mei-Chen Hsueh, Timothy K. Tsai und Ravishankar K. Iyer. *Fault Injection Techniques and Tools*. Techn. Ber. University of Illinois at Urbana-Champaign, 1997.

[Imb] *Imbus.de Testbegriffe Glossar*. URL: http://www.nmea.de/nmea0183datensaetze.html.

[JMF98] J.R.Samson, Jr. W. Moreno und F. Falquez. *A technique for automated Validation of fault tolerant ddesign using laser fault Injection (LFI)*. Techn. Ber. Proceedings of 27th International Symposium on Fault Tolerant Computing (München), 1998.

[Jta] *http://en.wikipedia.org/wiki/Image:Jtag_chain.svg.*

[Juh03] Thomas Juhnke. "Die Soft-Error-Rate von Submikrometer-CMOS-Logikschaltungen". Magisterarb. Technische Universität Berlin, 2003.

[Kem00] G. Kemnitz. "Test und Verlässlichkeit von Rechnern". In: *TU Clausthal (Institut für Informatik)* (2000).

[Klo12] Erich Klock. *Beschlusslage der EU Weltraumpolitik und ihre Bedeutung für Österreich.* Techn. Ber. ESPI - European Space Policy Institute, 2012.

[Mag07] Phrack Magazine. *Low Cost and Portable GPS Jammer.* 2007. URL: http://www.phrack.org/issues.html?issue=60&id=13#article.

[Mar03] Richard P. Martin. *Heisenbugs and Bohrbugs: Why are they different?* Techn. Ber. Rutgers University, 2003.

[Mil] *http://embdev.net/topic/235605.*

[Mil72] Harlan D. Mills. *On the Statistical Validation of Computer Programms, Technical Report FSC-72-6015.* Techn. Ber. Gaithersburg MD: IBM Federal Systems Division, 1972.

[MOK05] Yu-Seung Ma, Jeff Offutt und Yong Rae Kwon. "MuJava: An Automated Class Mutation System". In: *Software Testing, Verification and Reliability* (2005), S. 97–133.

[Moo05] Ivan Moore. *Jester - the JUnit test tester.* 2005. URL: http://jester.sourceforge.net/.

[Mye99] Glenford J. Myers. *Methodisches Testen von Programmen.* R. Oldenbourg Verlag München Wien, 1999.

[Möc12] Erich Möchel. *Achillesferse GPS-System.* 2012. URL: http://fm4.orf.at/stories/1701529/.

[Nmea] *NMEA-Datensätze.* URL: http://www.nmea.de/nmea0183datensaetze.html.

[Nmeb] *NMEA-Website:Online.* URL: http://www.nmea.org/content/nmea_standards/nmea_0183_v_410.asp.

[Psp] *Cadence PSpice A/D and Advanced.* URL: http://www.cadence.com/products/orcad/pspice_simulation/Pages/default.aspx.

[Rey13] Victor Reyes. "Virtual Hardware In-the-Loop: Earlier Testing for Automotive Applications". In: *Whitepaper der Firma Synopsys* (2013).

[Ric13] Arne Maximilian Richter. "Konzept und Einführung von Safety-Analysen bei Mikrocontroller-basierten Anwendungen in UAVs". Magisterarb. University of Applied Science Hamburg, 2013.

[Rie08] Ger Rietman. *How to calculate the NMEA checksum*. 2008. URL: http://rietman.wordpress.com/2008/09/25/how-to-calculate-the-nmea-checksum/.

[RLS78] R.A.DeMillo, R.J. Lipton und F.G. Sayward. "The design of a prototype mutation system for program testing". In: *Conference Record* (1978), S. 623–627.

[RS09] J. Reichardt und B. Schwarz. *VHDL-Synthese*. Oldenbourg, 2009.

[Sch10a] Prof. Dr. Dr.h.c.mult. August-Wilhelm Scheer. *Eingebettete Systeme - Ein strategisches Wachstumsfeld für Deutschland*. BITKOM - Bundesverband Informationswirschaft, Telekommunikatuon und neue Medien e.B., 2010.

[Sch10b] Rüdiger Schobbert. *Ausgabekünstler - Logging mit SLF4J/Logback*. 2010.

[SL10] Andreas Spillner und Tilo Linz. *Basiswissen Softwaretest*. 4. übberarbeitete und aktualisierte Auflage. dpunkt.verlag, 2010.

[Sof] *Soft Errors in Electronic Memory - A White Paper*. URL: http://www.tezzaron.com/about/papers/soft_errors_1_1_secure.pdf.

[Som07] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishers Limited, Pearson Education Limited, 2007.

[Sto96] Neil Storey. *Safety Critical Computer Systems*. Addison Wesley Pub Co Inc, 1996.

[TB] O. Tschäche und F. Balbach. *VHDL-based Fault Injection with VERIFY*. Techn. Ber. Universität Erlangen.

[Tho00] Robert Thorhuus. "Software Fault Injection Testing". Magisterarb. Electronic System Design - University of Stockholm, 2000.

[TW79] T.C.May und M.H. Woods. "Alpha-Particle-Induced Soft EError in Dynamic Memories". In: *IEEIEEE Transactions on Electron Devices-26* (1979), S. 2–9.

[Ull11] Christian Ullenboom. *Java ist auch eine Insel*. Galileo Computing, 2011.

[VK03] Uwe Vigenschow und Dietmar Kropftisch. "Das Fehlermodell: Aufwandschätzung und Planung von Tests". In: *OOSE/www.objektspektrum.de* (2003).

[VM98] Jeffrey M. Voas und Gary McGraw. *Software Fault Injection*. Wiley Computer Publishing, 1998.

[Wil13] Laurie Williams. *muJava Home Page*. 2013. URL: http://cs.gmu.edu/~offutt/mujava/.

[WJ03] Jon S. Warner und Roger G. Johnston. "GPS Spoofing Countermeasures". In: *Homelandsecurity Journal* (2003).

[Xce] *XCeption - Automated Fault-Injection Environment (Flyer) der Firma Critical Software Limited.*

Glossar

AES	Airborne Embedded Systems
BWB	Blended Wing Body
GGA	Global Positioning System Fix Data
RMC	Recommended Minimum Sentence C
FSM	Final State Machine
FCU	Flight Control Unit
FDL	Flugdaten Logger
LDS	Luftdaten Sammler
PWM	Pulsweitenmodulation
NMEA	National Marine Electronics Association
FPGA	Field Programmable Gate Array
DI	Dependency Injection
HWIFI	Hardware Implemented Fault-Injection
SWFI	Software Fault-Injection
SFI	Simulated Fault-Injection
VHDL	Very High Speed Integrated Circuit Hardware Description Language
vHIL	Virtual Hardware in the Loop
DIN	Deutsches Institut für Normung
GPS	Global Positioning System
SLF4J	Simple Logging Facade for Java
DRAM	Dynamic Random Access Memory
ASCII	American Standard Code for Information Interchange

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 14. April 2014	
Ort, Datum	Benjamin-Yves Johannes Trapp