

Verlässliche Echtzeitsysteme

Fehlerinjektion

Peter Ulbrich

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.informatik.uni-erlangen.de

16. Juni 2015



Fragestellungen



- Extern verursachte Fehler sind die (**absolute**) **Ausnahme**!
- Ausfallrate \ll Überlebensfunktion (vgl. auch III/12)
 - Nachweis der **Wirksamkeit von Fehlertoleranzmechanismen?**



- Dedizierte **Testmethoden** sind vonnöten
- Fehlertoleranzmechanismen „verarbeiten Fehler“
 - Man braucht entsprechend Fehler, um diese Mechanismen zu testen
 - Konkrete Umsetzung ist jedoch aufwendig ...
 - Fehler (auch „häufige“ transiente Fehler) lassen sich **nicht einfach abwarten**
 - Fehler verursachen mitunter **sehr hohe Kosten**

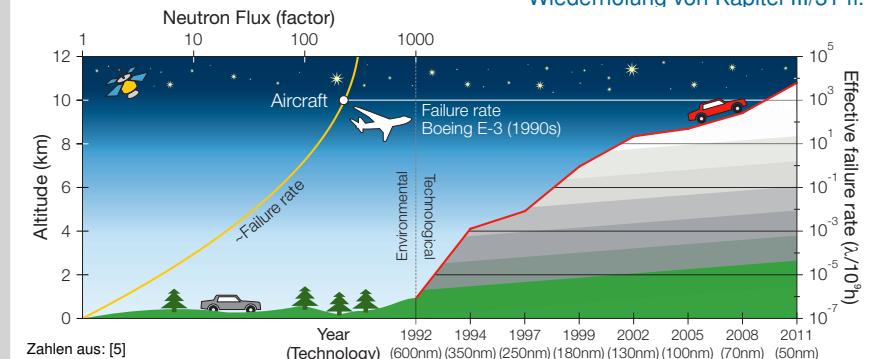


- Fehlerinjektion** als Mittel der Wahl
- Gezielte und reproduzierbare Erzeugung von Fehlern
 - **Validierung** von Fehlertoleranzmechanismen
 - **Bewertung** von Fehlertoleranz
 - inhärente Robustheit, Fehlerausbreitung, Fehlererkennungslatenz und -rate



Fehlerraten transienter Fehler

Wiederholung von Kapitel III/31 ff.



- Bitkipper durch **Umladungen in Speicherzellen und Schaltkreisen**
 - Verursacht durch **ionisierende Strahlung**
 - Erzeugung von Elektron-Loch-Paaren im Halbleitermaterial
- Rauschen durch **elektromagnetische Interferenz**
 - Verfälschung von **Kommunikation auf Bussen**
 - z. B. in Automobilen gibt es verschiedene Quellen für Wechselfelder



Beispiel: Problemstellung

Fehlerinjektion gibt es nicht nur bei Computern

- Moderne Automobile umfassen eine Vielzahl von Schutzsystemen
 - Air-Bag (Fahrer, Beifahrer, ...), Seitenauftprallschutz, Gurtstraffer, ...
 - Frage: Wie wirksam sind diese Systeme?
- Daten aus dem täglichen Betrieb von Autos mit realen Unfällen ...
 - ... sind **nicht ausreichend vorhanden** (eher seltene Unfälle)
 - ... sind **viel zu teuer** (Verlust von Menschenleben inakzeptabel)



Fehlerinjektion durch Crashtests



Gliederung

1 Überblick

2 FARM

3 Fehlerinjektionstechniken

4 FAIL*

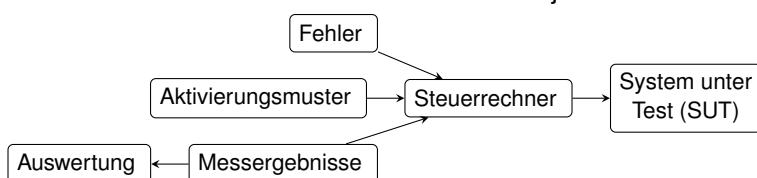
5 Zusammenfassung



Fehlerinjektion – Was braucht man da alles? (Forts.)

Irgendwie muss sich das auch noch bewegen ...

- **Auswahl** des zu injizierenden Fehlers
 - Unterschiedliche Prüfstände für Frontal- bzw. Seitenaufprall
 - **Ausführung** des Aktivierungsmusters
 - Beschleunigung des Fahrzeugs auf die gewünschte Geschwindigkeit
 - **Beobachtung** der Folgen der Fehler situation
 - Sensoren erfassen Beschleunigungen, Verwindungen, Verformungen, ...
 - **Auswertung** der Messergebnisse
 - Abgleich mit á-priori Wissen ~ Schluss auf Verletzungen
- ein **Steuerrechner** übernimmt i. d. R. die Fehlerinjektion in Rechner



Fehlerinjektion – Was braucht man da alles?

- Erstmals festgelegt im **FARM-Modell** [2]
 - Anmerkungen beziehen sich auf Crashtests (s. Folie 4)

Fault ~> Fehlerraum

- Frontal- oder Seitenaufprall, Geschwindigkeit, ...
- Bezieht sich auf eine **realistische Fehlerhypothese**

Activation ~> Aktivierungsmuster

- Das beschleunigte Auto fährt auf den Prellbock zu
- Das **Auftreten des Fehler** wird herbeigeführt

Readout ~> Messergebnisse

- Deformierung der Fahrgastzelle, ...
- Erhebung der **beobachtbaren Folgen des Fehlers**

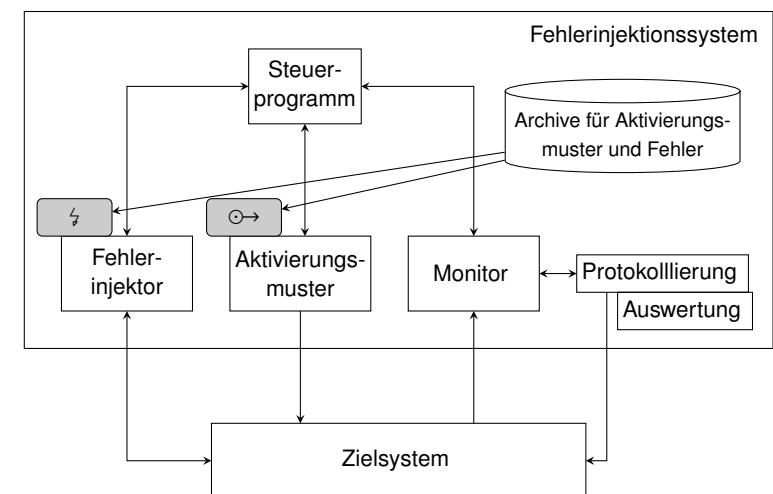
Measure ~> Bewertung der Messergebnisse

- Insassen würde schwere innere Verletzungen erleiden
- Wie **zuverlässig** ist mein System?



Genereller Aufbau von Fehlerinjektionswerkzeugen

Quelle Grafik: [3]



Ablauf einer Fehlerinjektion

- Fehlerinjektion besteht aus Experimenten:
 - 1 Der Steuerrechner wählt
 - Einen Fehler f aus dem Fehlerraum F und
 - Ein Aktivierungsmuster a aus der Menge der Aktivierungsmuster A
 - 2 Anschließend wird die Fehlerinjektion durchgeführt
 - Starten des Aktivierungsmusters a
 - Injizieren des Fehlers f
 - 3 Abschließend werden die Messergebnisse r erfasst
 - Jedes Experiment wird durch einen Tupel (f, a, r) beschrieben
- ☞ Gesamtheit der Messergebnisse $R \sim$ Zuverlässigkeitmaße
 - Fehlererkennungslatenz- und -rate, Erholungszeit, ...
- ⚠ Eine Kampagne (engl. campaign) beinhaltet mehrerer Experimente
 - Der Fehlerraum ist meist sehr groß \sim eine Vielzahl von Fehlern
 - Hinzu kommen die Möglichkeiten für ihre Aktivierung



Fehlerraum

- Transiente Fehler haben ihren Ursprung im physikalischen Modell
 - Umwelteinflüsse bewirken Zustands-/Ladungsveränderungen
 - Spannungsschwankungen bei der Datenübertragung
 - Umladen von der Kondensatoren in Speicherzellen (EMV, Strahlung)
 - Veränderung der Leitfähigkeit durch Hitzeeinwirkung
 - Fehlerhafte Berechnungen in arithmetischen Einheiten
 - Annahme: diese Veränderungen werden in der Software sichtbar
- ☞ Als „Bitkipper“ im empirischen Modell beobachtbar
 - In der Halbleitertechnik der Hardware
 - Hierfür existieren sehr, sehr viele Möglichkeiten
- ☞ Wir interessieren uns für in Software sichtbare „Bitkipper“
 - Register, Speicher, Instruktionen, ...
 - Woher sie kommen, ist nicht so sehr von Belang!



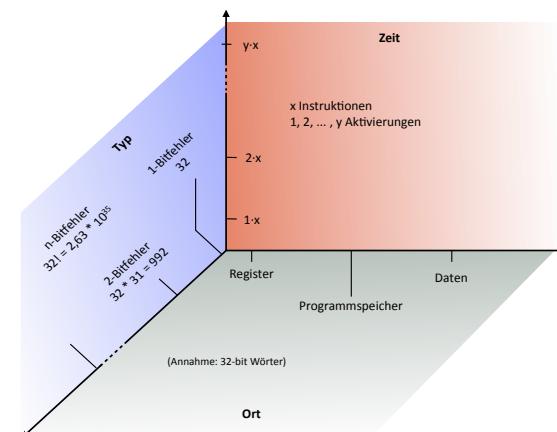
Fehlerinjektion – Abstraktionsebenen

- Fehler können auf verschiedenen Ebenen injiziert werden [1]
 - ☞ Axiomatische Modelle
 - Analytische Modelle bilden das Verhalten des Systems ab
 - Markov-Ketten, Petri-Netze, Zuverlässigkeitssblockdiagramme
 - ☞ Empirische Modelle
 - Detailliertere Modelle für Systemverhalten und -struktur
 - Erfordern i. d. R. simulationsbasierte Ansätze
 - ☞ Physikalische Modelle
 - Reale Implementierung des Systems in Hard- und/oder Software
- ⚠ Wahl der Ebene hat signifikanten Einfluss auf die Fehlerinjektion
 - Insbesondere die Mengen F und A hängen von ihnen ab
 - Fehlerhypothese: durch Software beobachtbare transiente Fehler
 - Konzentration auf empirische/physikalische Modelle



Fehlerraum (Forts.)

Ein Musterbeispiel für eine kombinatorische Explosion



- ⚠ Selbst 1-Bitfehler spannen einen dramatisch großen Fehlerraum auf!
 - In welchem Register möchte man ein Bit kippen lassen?
 - Nach welcher Instruktion soll das Bit gekippt werden?



Aktivierungsmuster

- Umfassen die Durchführung der zu schützenden Berechnung
 - Einfachster Fall: Vektoren von Eingabeparametern
 - i. d. R. nur für einfache Soft- oder Hardwareimplementierungen anwendbar
 - Präparation **anwendungsspezifischer, passender Eingabedaten**
 - Sensordaten, Netzwerkpakete, ...
 - Kombinationen aus Soft- und Hardware → Kontrollfluss
 - Unterbrechungen werden hier zum Problem
 - Echtzeitsysteme erfordern eine **Umgebungssimulation** (s. IV/33 ff.)
 - Durchführung am „realen Objekt“ häufig nicht möglich/zu gefährlich
 - Eingaben müssen das Verhalten des physikalischen Objekts widerspiegeln
- ☞ Referenzlauf (engl. **golden run**) liefert das gewünschte Verhalten
 - Bestimmung des Ergebnisses ohne Fehlerinjektion
 - Aufzeichnung des Ein-/Ausgabeverhaltens des SUT
 - Dient dem späteren Abgleich und der Erkennung von SDCs
- ☞ Anschließend folgt die **eigentliche Fehlerinjektion**
 - Einbringen des gewünschten Fehlers



Physikalisches vs. empirisches Modell

Besteht ein Zusammenhang zwischen diesen beiden Welten?

- ⚠ Vereinfachung: In Software sichtbare Fehler → **Bitkipper**
 - Ihr Zustandekommen ist uninteressant
 - Verzicht auf eine indirekte physikalische Fehlerinjektion
 - Diese ist **sehr aufwendig** und **schlecht kontrollierbar**
 - Benötigt beispielsweise EMV-Messkammern oder Ionen-Kanonen
 - Welches Bit in welchem Register wurde denn nun beeinflusst?
- ☞ Fehlerinjektion bringen Bitkipper direkt ein
 - Keine direkte, physikalische Manipulation notwendig
 - **Simulation von Fehlern auf Registertransferebene**
 - Nicht zu verwechseln mit **Fehlersimulation** (engl. *fault simulation*)
 - Hier wird ein Schaltkreis in Anwesenheit von Fehlern simuliert
 - Man spricht von: **Software Implemented Fault Injection** (SWIFI)
 - Falls eine Softwareimplementierung die Verfälschung durchführt
 - Alternativ: Verwendung spezieller Debug-Schnittstellen (z. B. JTAG)
 - **Scan-Chain Implemented Fault Injection** (SCIFI)



Messergebnisse und ihre Bewertung

- Mächtigkeit des Zielsystems bestimmt erfassbare Messergebnisse
 - ☞ Hilfreich sind folgende Informationen
 - **Fehlerparameter**
 - Wann und wo wurde der Fehler injiziert? Welcher Typ wurde injiziert?
 - **Systemkontext**
 - Werte der Register, Auszug eines Speicherbereichs
 - Was hat der Fehler verändert? Wie hat er sich fortgepflanzt?
 - **Rückgabewerte, Rechenergebnisse**
 - Hat die Fehlerinjektion die Berechnung beeinflusst?
 - **Ausführungszeit**
 - Wie lange dauert es bis der Fehler aktiviert, entdeckt oder maskiert wurde?
 - **Fehlererkennungsmechanismen**
 - Welcher Fehlerdetektor schlug an?
 - ☞ Hieraus werden **Maße zur Beurteilung der Fehlertoleranz** bestimmt
 - Rate der Fehlererkennung und Maskierung, Latenz, Erholungszeit



Gliederung

- 1 Überblick
- 2 FARM
- 3 Fehlerinjektionstechniken
- 4 FAIL*
- 5 Zusammenfassung



Fehlerinjektionstechniken

- Injektion von Fehler auf allen Ebenen eines Rechensystems möglich
- ☞ es existiert eine Vielzahl verschiedener Techniken [6]
 - **Hardware-basierte** Techniken
 - Integriert spezialisierte Hardware in das zu testende System
 - **Software-basierte** Techniken
 - Modifiziert die zu testende Software, um fehlerhaftes Verhalten zu erzeugen
 - **Simulations-basierte** Techniken
 - Simulation des zu testenden Systems, basierend z. B. auf VHDL
 - **Emulations-basierte** Techniken
 - Beschleunigt Simulation durch Synthesierung des Modells
 - **Hybride Ansätze**
 - Vereinigt zwei oder mehr der oben genannten Ansätze



Hardware-basierte Fehlerinjektion

- Hardware-basierte Implementierung ~ „Simulation“
 - Enthaltene Testschaltungen injizieren direkt transiente Fehler
 - Basiert auf dem komplett gefertigten Schaltkreis
 - Gefertigter und getester Schaltkreis verhalten sich identisch
 - Das Verfahren ist **nicht-intrusiv** (engl. *non-intrusive*)

Hierbei stehen folgende Möglichkeiten offen:

- **Mit Kontakt** ~ direkte Manipulation elektrischer Signale
 - Anbringungen aktiver Messfühler an einzelnen Prozessorspins
 - Hängen gebliebene Signale (engl. *stuck-at, stuck-open*)
 - Überbrückung mehrerer Signale (engl. *bridging*)
 - Verwendung von **Zwischensockeln** (engl. *socket insertion*)
 - Implementierung beliebiger Funktionen auf den eingehenden Signalen
- **Ohne Kontakt** ~ indirekte Manipulation elektrischer Signale
 - Der Schaltkreis wird physikalischen Phänomenen ausgesetzt
 - Radioaktive Strahlung, elektromagnetische Interferenz, Hitze, ...
 - Rufen (relativ unkontrolliert) transiente/permanente Fehler hervor



Hardware-basierte Fehlerinjektion (Forts.)

Vorteile

- + Hohe zeitliche Auflösung der Injektion und Beobachtung
 - Ermöglicht akkurate Aussagen zu Fehlererkennungsrate und -latenz
- + Unterstützt nicht-intrusive Fehlerinjektion
 - Betrachtet das komplette System, sowohl Soft- als auch Hardware
- + Durchführung der Experimente ist sehr schnell

Nachteile

- Eine Beschädigung des getesteten Systems ist möglich
- Hohe Integrationsdichten erschweren die Fehlerinjektion
- Erfordert spezielle Hardware ~ geringe Portierbarkeit
- Eingeschränkte Kontrollier- und Beobachtbarkeit
 - Nur bestimmte Fehlertypen sind injizierbar
 - Nicht alle Stellen des Schaltkreises sind direkt zugänglich



Software-basierte Fehlerinjektion

- Spezielle Softwarekomponenten übernehmen die Fehlerinjektion
- **Zur Übersetzungszeit** (engl. *compile-time*)
 - Wird das Programmabbild verändert, bevor es geladen wird
 - Für die Fehlerinjektion werden gezielt Software-Defekte eingebracht
 - Die eigentliche Fehlerinjektion ist also die Erzeugung des Abbilds
 - Ausführung des Abbilds aktiviert die eingefügten Defekte
 - Diese simulieren transiente/permanente Hard-/Softwarefehler
- **Zur Laufzeit** (engl. *run-time*)
 - Erfordert die Aktivierung des Fehlerinjektionsmechanismus
 - z. B. durch **Auszeiten**, **Traps** oder **Instrumentierung**
 - Die Behandlung der Ereignisse führt die Fehlerinjektion durch
 - Instrumentierung bereitet die Fehlerinjektion vor
 - Bringt gezielt Instruktionen in das Programmabbild ein
 - Diese aktivieren dann die Fehlerinjektion



Vorteile

- + Sehr flexible Injektion von Fehlern möglich
 - Fehler in Registern, Speicher, bei der Kommunikation, im Zeitbereich
 - Injektion ist in Simulationen und realen Systemen möglich
- + Durchführung der Experimente ist sehr schnell
- + Keine Spezialhardware erforderlich

Nachteile

- Eingeschränkte Auswahl von Injektionsstellen
 - i. d. R. auf der Ebene von Assemblerinstruktionen
- Eingeschränkte Kontrollier- und Beobachtbarkeit
- Erfordert eine Modifikation der getesteten Software
 - Letztendlich wird ein anderes Programmabbild verwendet
 - Injektionsverfahren ist intrusiv → es beeinflusst das Verhalten



- Ein Modell des zu testenden Systems wird im Simulator ausgeführt
 - Das Modell umfasst z. B. Prozessor, Peripherie, Kommunikation, ...
- Fehlerinjektion basiert auf:
 - **Modifikation des Systemmodells** → vgl. software-basierte Lösung
 - **Saboteure**: „boshaft“ in das Modell eingebrachte Komponenten
 - Aktivierung → Ausführung der Fehlerinjektion (z. B. Signalstörung)
 - Ansonsten verhalten sie sich unauffällig
 - **Mutanten**: „boshaft“ veränderte Komponenten des Modells
 - Veränderte, existierende Komponenten injizieren Fehler
 - **Modifikation der Simulation** → vgl. hardware-basierte Lösung
 - Erfordert keine Veränderung des Modells sondern des Simulators
 - Injektion von Fehlern an beliebigen Stellen/Zeitpunkten
 - Modifikation von Zuständen oder Signalen



Vorteile

- + Größtmögliche Flexibilität: Abstraktionsebene/Fehlerhypothese
 - Auf elektrischer, logischer, funktionaler und architektureller Ebene
 - Injektion zeitlicher, transienter und permanenter Fehler
- + Nicht-intrusive Injektion möglich (unverändertes Programmabbild)
- + Erfordert keine Spezialhardware
- + Maximaler Grad an Kontrollier- und Beobachtbarkeit

Nachteile

- Hoher Zeitaufwand
 - Erfordert die Entwicklung von (detaillierten) Systemmodellen
 - Die Simulationsgeschwindigkeit ist häufig niedrig
- Hängt von der Akkuratheit des Systemmodells ab
 - Kein 100%-iges Abbild der Realität (→ keine „Echtzeitsimulation“)



- 1 Überblick
- 2 FARM
- 3 Fehlerinjektionstechniken
- 4 FAIL*
- 5 Zusammenfassung



Fehlerinjektion: Praktische Probleme

- Validierung von CoRed (s. IX/34 ff.) durch Fehlerinjektion
 - Erster Ansatz: Debug-Schnittstelle des TriCore (OCDS)
 - Steuerrechner: Debugger Trace 32 von Lauterbach
 - Skriptgesteuerte Ausführung von Ausführung, Injektion und Protokollierung

☞ Durchführung von Fehlerinjektion ist eine **große Herausforderung**

- Man kämpft mit einem **riesigen Fehlerraum**
- Experimentbeschreibung ist **nicht standardisiert/wiederverwendbar**
- **Hoher zeitlicher Aufwand:** 1s/Experiment * 400.000 Experimente \sim 110 h

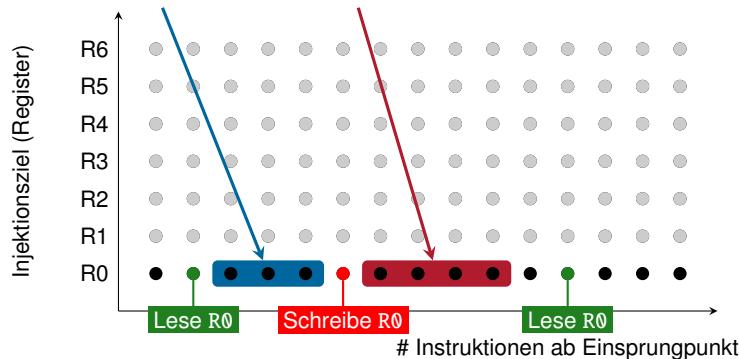
⚠ Prinzipiell existiert eine Vielzahl von Werkzeugen, aber:

- Diese sind **hochgradig proprietär**
 - Eigene Fehlermodelle, Experimentbeschreibung, Ergebnisauswertung
- An **bestimmte Zielplattformen** gebunden
 - Erweitern häufig (veraltete Versionen) existierender Emulatoren
- Eine einfache Verwendung „out-of-the-box“ unmöglich



Reduktion der Kampagnendauer

- Reduktion des Fehlerraums durch „**fault-space pruning**“
 - Register R1 ... R6 sind uninteressant
 - Eliminiere **unwirsame** und **idempotente** Injektionen



- einzelne Experimente sind **unabhängig voneinander**
 - Sie lassen sich **hervorragend parallelisieren**
 - Auf mehreren Kernen, Prozessoren, Rechnern, ... in der Cloud



Lösungsansatz: FAIL* [4]

- FAIL* \leadsto **Fault Injection Leveraged**
 - Vorrangiges Entwurfziel: Flexibilität bei der Fehlerinjektion

☞ Verwendung **existierender virtueller Plattformen** (Bochs, OVP, ...)

- Aktuelle, gewartete Softwarebasis
- Schneller Wirtsrechner \leadsto schnelle Durchführung von Experimenten
- Voller Zugriff auf und volle Kontrolle über die Plattform

☞ Schaffung einer **abstrakten Schnittstelle** zu diesen Plattformen

- Wiederverwendbare Beschreibung von Experimenten

⚠ Mehr dazu in den Übungen!

Gliederung

- 1 Überblick
- 2 FARM
- 3 Fehlerinjektionstechniken
- 4 FAIL*
- 5 Zusammenfassung



FARM-Modell Für Fehlerinjektion

- Fault, Activation, Readout, Measure
- Auswahl, Ausführung, Beobachtung, Auswertung
- Abstraktionsebenen – axiomatisch, empirisch, physikalisch
- genereller Aufbau und Ablauf von Fehlerinjektionswerkzeugen

Fehlerinjektionstechniken → grundlegende Kategorisierung

- {hardware, software, simulations, emulations}-basiert

FAIL* → Grundlage für generische Fehlerinjektion?

- Basierend auf virtuellen Zielsystemen
- flexible Plattform für Fehlerinjektion
- schnelle Experimentdurchführung durch Parallelisierung



- [1] ARLAT, J. ; CROUZET, Y. ; LAPRIE, J.-C. :
Fault injection for dependability validation of fault-tolerant computing systems.
In: *Proceedings of the 19th International Symposium on Fault-Tolerant Computing (FTCS-19)*, 1989, S. 348–355
- [2] ARLAT, J. ; AGUERA, M. ; AMAT, L. ; CROUZET, Y. ; FABRE, J.-C. ; LAPRIE, J.-C. ; MARTINS, E. ; POWELL, D. :
Fault Injection for Dependability Validation: A Methodology and Some Applications.
In: *IEEE Transactions on Software Engineering* 16 (1990), Febr., Nr. 2, S. 166–182.
<http://dx.doi.org/10.1109/32.44380>. –
DOI 10.1109/32.44380. –
ISSN 0098-5589
- [3] HSUEH, M.-C. ; TSAI, T. K. ; IYER, R. K. :
Fault Injection Techniques and Tools.
In: *IEEE Computer* 30 (1997), Apr., Nr. 4, S. 75–82.
<http://dx.doi.org/10.1109/2.585157>. –
DOI 10.1109/2.585157. –
ISSN 0018-9162



Literaturverzeichnis (Forts.)

- [4] SCHIRMEIER, H. ; HOFFMANN, M. ; KAPITZA, R. ; LOHMANN, D. ; SPINCKY, O. :
FAIL*: Towards a Versatile Fault-Injection Experiment Framework.
In: MÜHL, G. (Hrsg.) ; RICHLING, J. (Hrsg.) ; HERKERSDORF, A. (Hrsg.): *25th International Conference on Architecture of Computing Systems (ARCS '12), Workshop Proceedings* Bd. 200, Gesellschaft für Informatik, März 2012 (Lecture Notes in Informatics). –
ISBN 978-3-88579-294-9, S. 201–210
- [5] SHIVAKUMAR, P. ; KISTLER, M. ; KECKLER, S. W. ; BURGER, D. ; ALVISI, L. :
Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic.
In: *Proceedings of the 32nd International Conference on Dependable Systems and Networks (DSN '02)*.
Washington, DC, USA : IEEE Computer Society Press, Jun. 2002, S. 389–398
- [6] ZIADE, H. ; AYOUBI, R. A. ; VELAZCO, R. :
A Survey on Fault Injection Techniques.
In: *The International Arab Journal of Information Technology* 1 (2004), Nr. 2, S. 171–186

