Zuverlässigkeitsprüfung eingebetteter Steuergeräte mit modellgetriebener Fehlerinjektion

Bernd-Holger Schlingloff[1,2], Saša Vulinović [3]

[1]Fraunhofer Institut
für Rechnerarchitektur und
Softwaretechnik FIRST
Kekulèstr. 7, 12489 Berlin
Holger.Schlingloff@first.fhg.de

[2]Humboldt-Univ. zu Berlin
Institut für Informatik
Rudower Chaussee 25,
12489 Berlin
hs@informatik.hu-berlin.de

[3]Technische Universität Berlin,
Institut für Technische Informatik
und Mikroelektronik
Franklinstr. 28/29, 10587 Berlin
vulinovi@cs.tu-berlin.de

Abstract: In dieser Arbeit beschreiben wir den Einsatz von Techniken der softwareimplementierten Fehlerinjektion in einer modellbasierten Entwicklungsumgebung. Die intendierte Klasse von Zielsystemen sind dabei verteilte Steuerungsfunktionen und vernetzte Steuergeräte im Automobil. Wir analysieren die auf den verschiedenen Ebenen möglichen Fehlerursachen und Ausfälle, identifizieren Fehlerklassen und Fehlermodelle für die Injektion, und skizzieren den Aufbau einer Simulationsumgebung für die Durchführung der Simulationsläufe.

1 Einführung

Die Entwicklung automobiler Steuergeräte erfolgt heute zunehmend modellbasiert. Auf der Basis von Matlab/Simulink®-Modellen wird mit Codegeneratoren C-Code oder Produktionscode für spezielle Prozessoren generiert, der die Funktionen des Steuergerätes realisiert. Die Beschreibung der Funktionen erfolgt dabei in mehreren Stufen: Die ursprünglichen Anforderungen werden in einem Funktionsmodell (*platform independent model*, PIM) beschrieben, welches schrittweise in ein Implementierungsmodell (*platform specific model*, PSM) überführt und mit speziellen Details für die Zielplattform angereichert wird. Die einzelnen Funktionen im Fahrzeug sind dabei mehr und mehr miteinander vernetzt und voneinander abhängig. Ein Trend ist daher die Entwicklung von Steuergeräte-übergreifenden Funktionsmodellen, bei denen die konkrete Verteilung der einzelnen (Sub-)Funktionen auf einzelne Steuergeräte erst sehr spät festgelegt wird. Ein Vorteil dieser Vorgehensweise liegt in der Möglichkeit, die Anzahl und Varianz der verwendeten Steuergeräte in den Griff zu bekommen. Andererseits ergeben sich dadurch Probleme mit der Ausbreitung von Fehlern, da der Ausfall oder teilweise Ausfall eines Gerätes (Hardware oder Software) unüberschaubare Folgen auf das Gesamtsystem Fahrzeug haben kann.

In dieser Arbeit stellen wir Ansätze zur Simulation der Auswirkung von Fehlern auf verschiedenen Stufen vor, die auf der Grundlage der softwareimplementierten Fehlerinjektion (Software implemented fault injection, SWIFI) beruhen. Durch geeignete Modifikationen im Modell und im generierten Code lassen sich Rückschlüsse auf das Ausfallverhalten des Gesamtsystems (Hardware und Software) bilden. Auf diese Weise lassen sich die Effekte des Ausfalls einzelner Komponenten bei einer gegebenen Verteilung der Funktionen ermitteln und mögliche Schwachstellen im Design eingrenzen..

2 Softwareimplementierte Fehlerinjektion

Zur Ermittlung der Auswirkungen von Fehlern in eingebetteten Systemen wird vor allem im Aerospace-Bereich oftmals die so genannte Software-Fehlerinjektion (SWIFI) verwendet (siehe [1,2,3,4]). Man streut dabei gezielt Fehler in das Zielsystem ein und kann so gezielte Rückschlüsse auf das Systemverhalten im Ausnahmezustand ermöglichen. Das System wird dabei mit einer großen Anzahl von unterschiedlichen Fällen konfrontiert, die systematisch alle möglichen Funktionsausfälle abdecken. Vor allem zur Evaluation der Zuverlässigkeit von Mikroprozessoren sind hier mehrere Werkzeuge verfügbar [5,6,7]. Die Realisierung der Fehlerinjektion auf der Basis von zusätzlicher Software geschieht aus mehreren Gründen: Im Gegensatz zu analytischen Methoden ist die experimentelle Vorgehensweise skalierbar und liefert statistisch belegbare Resultate. Im Gegensatz zu physikalischen Fehlerinjektionen, z.B. durch Bestrahlung, Fehlerspannungen und andere physikalische Belastungen sind die Ergebnisse reproduzierbar und zerstörungsfrei.

Abhängig vom verwendeten Fehlermodell kann durch SWIFI einerseits der Effekt von systematischen (Software-) Fehlern simuliert werden. Andererseits ist es durch einen entsprechenden Detaillierungsgrad auch möglich, zufällige physikalische Fehler zu untersuchen.

3 Fehler auf Modell- und Code-Ebene

Wie oben dargelegt, wird auf der Modellebene zwischen funktionalem Modell und Implementierungsmodell unterschieden [8]. Auf der funktionalen Ebene werden abstrakte Funktionseinheiten, wie zum Beispiel ABS, ACC, Fahrdynamik-Regelung und ähnliches beschrieben. Im Implementierungsmodell ist die Verteilung der Funktionen auf die konkreten Steuergeräte sowie die algorithmische Realisierung festgelegt. Demzufolge können auf diesen Ebenen verschiedene Fehlermodelle angewendet werden.

Auf funktionaler Ebene lässt sich der Effekt von Sensor oder Aktuatorausfällen sowie fehlerhafte Kommunikation zwischen den einzelnen Funktionen modellieren. Hier können Werkzeuge wie CAPE/C® [9] eingesetzt werden mit denen sich interaktiv einzelne Elemente eines Funktionsmodells als fehlerhaft markieren lassen. Das Werkzeug berechnet dann die Auswirkungen dieses Defekts auf die übrigen Komponenten. Diese Berechnung kann als Basis für eine systematische Überprüfung dienen, bei der ein genau definierter Fehlerraum durchsucht wird. Die Festlegung der Korrektheitsbedingungen geschieht dabei ebenfalls auf funktionaler Modellebene. Eine Zusicherung besteht aus der Verbindung von Funktionseinheiten welche an einem spezifischen Fehlzustand beteiligt sein können und deren logischen Zusammenhängen.

Ein darüber hinausgehender Gedanke ist es, die Zusicherungen bei der Auswahl der zu injizierenden Fehler zu verwenden. Dabei wird die Fehlerinjektionsauswahl auf diejenigen Komponenten zu erweitern, welche transitiv von den direkt verbundenen Funktionseinheiten abhängen.

Bei der modellgetriebenen Fehlerinjektion auf der Ebene des Implementierungsmodells besteht die Möglichkeit, neue Fehlerursachen zu berücksichtigen. Wir unterscheiden hier zwischen Mutanten und Saboteuren. Mutanten entstehen dadurch, dass ein Block (z.B. eine Konstante oder ein Vergleich) durch einen ähnlichen Block ersetzt wird. Bei Saboteuren wird das Modell um zusätzliche Blöcke (z.B. Jitter auf Konstante) erweitert.

In [10] werden folgende Sabotagestrategien vorgeschlagen:

- **Break Line**
  Unterbrechen einer Linie zwischen zwei Blöcken. Der Ausgangsblock wird mit einem Terminator abgeschlossen und der Zielblock mit einer Konstanten (meist 0) versorgt.
- **Noise/Chirp**
  Zwischen zwei Blöcken wird eine Störung aufgeschaltet. Die Störung besteht aus einer Tabelle mit Impulsen, intermittierenden Unterbrechungen oder einem Summierer.
- **Gain**
  Das Signal zwischen zwei Blöcken wird um einen einstellbaren Faktor verstärkt.

Als Mutationsstrategien können folgende verwendet werden:

- **Mixing Lines**
  (Paarweises) Vertauschen von Ausgangs- oder Zielpunkten von Verbindungen
- **Random Selection**
  Ersetzung eines beliebigen Blocks durch einen anderen gemäß einer vorgegebenen Liste

Da für die Auswahl der zu injizierenden Fehler im Implementierungsmodell erheblich größere Freiheitsgrade bestehen als im Funktionsmodell, ist hierfür eine fundierte Strategie an Hand eines detaillierten Fehlermodells unerlässlich. Das zu verwendende Fehlermodell muss dabei *Betriebstreue* und *statistische Unabhängigkeit* gewährleisten; d.h., die Verteilung der injizierten Fehler muss realistisch und zufällig sein. Wenn diese Bedingungen eingehalten werden, lässt sich als Ergebnis einer großen Zahl von Simulationsläufen ein numerischer Wert für die Zuverlässigkeit berechnen [11]. Es ist auch möglich, die fehlerbehafteten Modelle zur Erstellung von Testfällen oder zur Evaluation von gegebenen Testsuiten zu verwenden [12, 13].

Ein wesentlicher Vorteil der modellbasierten Entwicklung besteht in der Möglichkeit der automatischen Codegenerierung mit Werkzeugen wie dSPACE TargetLink® oder MathWorks Real Time Workshop®. Die Stärke dieser Werkzeuge besteht dabei in einer hochgradigen Optimierung des generierten Codes für verschiedene Zielplattformen. Diese Optimierung erschwert aber zugleich die Verfolgung bzw. Rückverfolgung von Fehlern und Ausfällen und beeinflusst die auf Modellebene erhaltenen Ergebnisse von Simulationen. Daher ist es empfehlenswert, zusätzlich eine Fehlerinjektion auf Code-ebene durchzuführen. In [11] haben wir dabei folgende Fehlerinjektionsmöglichkeiten im Assembler identifiziert:

- Ausführung einer fehlerhaften Operation
- Manipulation von Operanden
- Verändern von Speicheradressen
- Modifikation von Registern

Um die Betriebstreue zu gewährleisten, wird dabei der zu untersuchende Code zunächst unter möglichst realistischen Belastungen ausgeführt und die tatsächliche Verwendung der Hardware-Ressourcen (Befehle, Speicherzellen, Register usw.) durch das entsprechende Programm gemessen. Die Strategie der Injektion richtet sich dann nach der statistischen Häufigkeit des Zugriffs auf diese Ressourcen. In einer Fehlerdatenbank wird eine Liste mit zu injizierenden Fehlern gebildet. Während der Durchführung der Simulation überwacht ein Debugger den Zugriff auf die Ressourcen und verändert in Abhängigkeit von der Fehlerdatenbank die entsprechenden Werte. Die Simulations-umgebung beobachtet nach Injektion des Fehlers das Verhalten des Systems und die Ausfallsymptome und protokolliert diese in der Fehlerdatenbank.

Es ist klar, dass die unterschiedliche Art von Fehlern, die auf den verschiedenen Stufen injiziert werden kann, zu ganz unterschiedlichen Ausfällen und Messergebnissen führt. Während auf der funktionalen Ebene vor allem der Ausfall ganzer Komponenten (Sensoren, Funktionen, Funktionsblöcke usw.) untersucht wird, stehen auf der Ebene des Implementierungsmodells vor allem algorithmische Aspekte im Vordergrund. Die Injektion von Fehlern im Zielcode simuliert dagegen eher zufällige physikalische Fehler in der Hardware und offenbart Schwachstellen im Zusammenspiel von Hard- und Soft-ware.

4. Co-Simulation

Die fortschreitende Integration von immer mehr Funktionen auf einem Steuergerät sowie die Weiterentwicklung der Prozessoren führen dazu, dass diese auch über immer mehr Rechenleistung verfügen. Technologisch sind Strukturgrößen von 65nm in naher Zukunft verfügbar. Dies erlaubt zum Beispiel hoch integrierte Multi-Core-Prozessoren in einem einzigen IC, welche die geforderte Rechenleistung zur Verfügung stellen können. Diese vergleichsweise kostengünstige Rechenleistung kann außerdem dazu verwendet werden, redundante Funktionseinheiten „On Chip“ vorzuhalten, welche die Aufgaben eines ausgefallenen Teils übernehmen, um dadurch die Zuverlässigkeit der Steuergeräte zu erhöhen und Wartungskosten zu senken. Wegen des technologischen Wandels wird dadurch eine Anpassung der Modelle nötig.

Für die bisher eingesetzte Halbleitertechnologie mit Strukturbreiten über 100nm ist eine Fehlerinjektion auf Codeebene ausreichend, um den Effekt von temporären Defekten (Single Event Upset, SEU) in einem Mikroprozessor zu modellieren. Die Studie in [14] bestätigt, dass man über 95% der temporären Single-Bit-Flips auf Register-Transfer Ebene durch Single-Bit-Flips auf Assemblerebene nachbilden kann. Diese Aussage beruht auf der Annahme, dass der größte Anteil an Fehlern aktueller Prozessoren in den Flipflops entsteht. Bei der neuen Technologie zeigen jedoch Studien wie [15,16], dass sich die Anzahl der Fehler in der Logik in die gleiche Größenordnung wie die Fehler in Flipflops bzw. DRAMs bewegen. Diese sind in den bisherigen Modellen vernachlässigt worden.

Eine Fehlerinjektion auf Gatter-Ebene ist technisch einfach durchzuführen. Jedoch ist auf Grund des relativ hohen Detaillierungsgrades und der insgesamt erhöhten Komplexität des Gesamtsystems mit langen Simulationszeiten zu rechnen. Daher ist es nötig, abstraktere Modelle zu entwickeln. Zur Lösung dieses Problems untersuchen wir den Zusammenhang zwischen den verschiedenen Modellebenen unter Berücksichtigung der neuen Technologie.

Im ersten Schritt lässt sich mit einer Systemsimulation in Verbindung mit einem Prozessormodell auf Gatterebene das Ausfallverhalten simulieren. Das Prozessormodell wird in einer Co-Simulation mit einem HDL Simulator (z.B. Modelsim®, [17]) simuliert und erzeugt dabei die Ausfälle welche auch im realen Betrieb zu erwarten sind. Diese Variante erfordert zwar hohen Zeit und Rechenaufwand zur Durchführung, wofür wir jedoch im Folgenden einige Ansätze zur Optimierung vorschlagen.

5. Implementierungsansatz

Ausgangspunkt unserer Simulationsumgebung ist das vom Designer erstellte funktionale Modell, welches detailliert genug ist, dass daraus optimierter C-Code generiert werden kann. Für die Fehlerinjektion wird der optimierte Code bei der Generierung zusätzlich so annotiert, dass Rückschlüsse auf die jeweiligen Modell-Blöcke gezogen werden können. Damit ist es möglich, jeden einzelnen Sub-Block mit einer gezielten Fehlerrate zu behandeln.

Um die Ergebnisse aus [14] für die 65nm Technologie zu übernehmen, erstellen wir ein Framework, welches als Werkzeug zur Fehlerinjektion auf Gatterebene und auf Codeebene dient. Es wird eine Beispielapplikation für die zu untersuchende CPU erstellt und eine Zuordnung von Programmzuständen zu den folgenden Kategorien vorgenommen:

- silent failure (latenter Fehler vorhanden, nicht maskiert)
- errorneous failure (Fehler offenbart sich)
- masked failure (Fehler ist maskiert worden)

Auf Code Ebene wird die Fehlerinjektion nach dem oben beschriebenen Verfahren angewendet und die Ergebnisse in die genannten Kategorien eingeordnet. Für die Gatter-Ebene wird die Fehlerinjektion mit einem HDL- Co-Simulator für einen beispielhaften Prozessor durchgeführt (z.B. LEON2, [18]). Auch hierbei werden Mutanten eingesetzt, welche die Funktionalität der zufällig ausgewählten Grundgatter ändern. Saboteure verändern alternativ die ursprünglichen Eingangssignale durch Addition von fehlerhaften Werten. Bei beiden Vorgehensweisen muss auf die Betriebstreue geachtet werden: Bei 65nm Technologien ist es z.B. erforderlich, dass Gatter und Flipflops zu je gleichen Teilen zu Fehlverhalten angeregt werden [19, 20]. Aus dem Vergleich der jeweiligen Injektions- und Ergebnisprofile beider Simulationen ergibt sich der Zusammenhang zwischen Code- und Assemblerebene bezogen auf das Ausfallverhalten von Software.

Die Art und Weise wie die Fehlerursachen ausgewählt werden, kann die Simulationsgeschwindigkeit stark beeinflussen. Eine Frage ist es dabei, zu klären, in wie weit Listen von Fehlerursachen auf Gatterebene minimiert werden können, wenn der Zustand aller Variablen a priori bekannt ist. Damit kann die Anzahl der nötigen Gesamtsimulationen begrenzt werden. Zu diesem Zweck ermitteln wir aus einer einmalig durchgeführten Referenzsimulation die Zustände aller im System vorkommenden Signale zu dem Zeitpunkt, an dem die Fehlerursache wirken soll. Dabei kann aufgrund der verwendeten Logik oft eine Voraussage getroffen werden, ob sich die Fehlerursache auf das Gesamtsystem auswirkt oder nicht.

Weiterhin kann die Gesamtsimulation beschleunigt werden, wenn nur an den Stellen Fehler injiziert werden, welche signifikant für den gerade ausgeführten Maschinenbefehl sind. Hier sind allerdings entsprechende Modelle von den Herstellern von Prozessoren nur schwer erhältlich. Für Prozessoren, für die eine Gatternetzliste oder eine Beschreibung auf Register-Transfer-Ebene erhältlich ist [z.B. ARM Core], kann so ein Modell vom Anwender selbst aufgestellt werden. Für einfache RISC Prozessoren sind wir hier wie folgt vorgegangen. Für jede vom Prozessor ausgeführte Instruktion und deren Varianten wird eine Liste aller Signalzustände gespeichert. Durch einen Kreuzvergleich wird die Gesamtwahrscheinlichkeit für den Signalzustand „1“ bzw. „0“ jedes Signals für den jeweiligen Befehl ermittelt und somit die Anzahl der notwendigen Simulationsläufe minimiert.

Der nächste Schritt ist die Loslösung von der Co-Simulation hin zur Erstellung eines abstrakten Modells. Dies kann z.B. ein Instruction Level Simulator sein, welcher z.B. als aktiver Block in Simulink® eingebunden wird und den HDL-Simulator ersetzt. Falls bereits ein änderbares Prozessormodell vorliegt, können auch der in Simulink® enthaltene Debugger zur Implementierung einer mutierten Blockkomponente einsetzt oder das Verhalten eines Saboteurs auf Code-Ebene nachgeahmt werden.

Mit beiden Möglichkeiten kann das Fehlverhalten des Originalprozessors statistisch betriebstreu nachgebildet werden, um damit eine Systemsimulation unter Simulink® zu ermöglichen. Das Fehlverhalten kann dann durch folgende Parameter gesteuert werden:

- Prozessormodell
- Prozessortechnologie
- Betriebsfrequenz
- Temperatur
- Alter

Abhängig von den Anforderungen der Anwendung kann so eine schnelle Systemsimulation durchgeführt werden, bei der das (Fehl-)Verhalten unter vorher festgelegen Randbedingungen bestimmt wird.

6. Zusammenfassung und Ausblick

Wir haben einen Überblick über Ansätze der modellgetriebenen Fehlerinjektion zur Ermittlung der Zuverlässigkeit vernetzter Steuergeräte im Automobil gegeben. Die Techniken der softwareimplementierten Fehlerinjektion können auf verschiedenen Ebenen der modellbasierten Entwicklung eingesetzt werden und erlauben demzufolge unterschiedliche Rückschlüsse auf das untersuchte System. Der qualitative und quantitative Zusammenhang der einzelnen Ausfallarten sowie die Korrelation zu Ausfällen auf der Hardwareebene sind allerdings noch nicht hinreichend geklärt und müssen weiter untersucht werden.

Insbesondere der Einfluss neuer Prozessortechnologien macht es notwendig, über eine Anpassung der vorhandenen Gate-Level Modelle nachzudenken. Änderungen im Low-Level-Modell ziehen Änderungen im High-Level Modell nach sich. Es wurden Ansätze präsentiert um die traditionellen Methoden weiter zu beschleunigen, sowie die neuen Parameter zu extrahieren und diese entweder im Rahmen einer HDL Co-Simulation des Prozessors oder mit Hilfe des Simulink® Debuggers in einer Systemsimulation zur Anwendung kommen zu lassen.

Zurzeit sind wir dabei, die beschriebenen Techniken zu implementieren und prototypisch anzuwenden. Ein wichtiger Punkt ist dabei der Vergleich von Fehlerursachen und Fehlerauswirkungen auf den verschiedenen Ebenen. Die Ergebnisse dieses Vergleichs können dazu beitragen, die verwendeten Fehlermodelle zu kalibrieren und somit die Allgemeingültigkeit der Methode zu erweitern.

Literaturverzeichnis

[1] J.-C. Fabre, F. Salles, M. Rodriguez-Moreno, J. Arlat: Assessment of COTS Microkernels by Fault Injection; Technical Report, LAAS-CNRS, Toulouse 2001

[2] M.-C. Hsueh, T. Tsai and R. Iyer, Fault Injection Techniques and Tools, Computer, vol. 30, pp. 75-82, 1997.

[3] J. Clark, D. Pradhan; Fault Injection: A method for Validating Computer-System Dependability, IEEE Computer, June 1995, pp. 47-56

[4] M.C.Calzarossa and S. Tucci (Eds.): *Measurement-Based Analysis of System Dependability Using Fault Injection and Field Failure Data.* Performance 2002, LNCS 2459, pp.290-317, 2002.

[5] Rodriguez, M., Salles, F., Fabre, J.C., Arlat, J.: MAFALDA: *Microkernel Assessment by Fault Injection and Design Aid*. Proceedings of the 3rd European Dependable Computing Conference (EDCC-3), 143–160, 1999

[6] J. Carreira, H. Madeira and J. G. Silva, Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers, IEEE Transaction on Software Engineering, vol. 24, pp. 125-136, 1998.

[7] G. A. Kanawati, N. A. Kanawati and J. A. Abraham, FERRARI: A Flexible Software-Based Fault and Error Injection System, IEEE Trans. on Computers, vol. 44, pp. 248-260, 1995.

[8] Potential and Challenges for Model-based Development in the Automotive Industry; in "Business Briefing: Global Automotive Manufacturing and Technology", World Market Research Center, Oktober 2000

[9] Capeware Technologies GmbH: Cape/C. http://www.capeware.de

[10] F. Geue: *Automatische Fault-Injection in SIMULINK Modelle;* Studienarbeit, Fachhochschule für Technik Esslingen, 2000

[11] S. Vulinovic und H. Schlingloff: Applikationsgeführte softwareinduzierte Fehlerinjektion eines fehlertoleranten Stellwerkscomputers; In: 16th ITG/GI/GMM

Workshop "Testmethoden und Zuverlässigkeit von Schaltungen und Systemen", FhG IAS/EAS, Dresden (Feb. 2004).

[12] S.G. Elbaum and J.C. Munson: Evaluating regression test suites based on their fault exposure capability; Journal of Software Maintenance: Research and Practice Volume 12, Issue 3 , Pages 171 - 184

[13] P.E. Ammann, P.E. Black, W. Majurski; Using Model Checking to Generate Test from Specifications; Proceedings of 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98), Brisbane, Australia (December 1998), edited by John Staples, Michael G. Hinchey, and Shaoying Liu, IEEE Computer Society, pages 46-54.

[14] Joakim Ohlsson: *On Concurrent Error Detection and Error Propagation,* Dissertation, Department of Computer Engineering, Göteburg 1995

[15] Alfredo Benso, Paolo Prinetto edt., "Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation", Kluwer Academic RPublishers,

[16] P. Shivakumar, "Modelling the Effect of Technology Trends on Soft Error Rate of Combinatorical Logic", Int. Conf on Dep Systems and Networks, June 2002

[17] Mentor Graphics: ModelSim. http://www.model.com/

[18] Gaisler Research: LEON2 Processor. http://www.gaisler.com/products/leon2/leon.html

[19] Assessing the Fault Tolerance of Embedded Software through Application of Machine Instruction Mutations, Proceedings of the IASTED International Conference on Modelling and Simulation MS'99 (4.- 8. Mai 1999, Philadelphia USA), S. 541-549. ISBN 0-88986-247-8

[20] Shekhar Borkar, Tanay Karnik, Vivek De: *Design and Reliability Challenges in Nanometer Technologies,* DAC 2004