Test und Fehlersuche in komplexen Autonomen Systemen*

Thomas Preusser[1], Albert Schulz[1], Alexander Weiss[1], Martin Heininger[2], Martin Leucker[3], Malte Schmitz[3], Torben Scheffel[3] und Daniel Thoma[3]

[1] Accemic Technologies GmbH
Franz-Huber-Str. 39, 83088 Kiefersfelden
{tpreusser,aschulz,aweiss}@accemic.com
[2] HEICON Global Engineering GmbH
Kreuzweg 22, 88477 Schwendi
martin.heininger@heicon-ulm.de
[3] Institut für Softwarettechnik und Programmiersprachen
Universität zu Lübeck
Ratzeburger Allee 160, 23562 Lübeck
{leucker,schmitz,scheffel,thoma}@isp.uni-luebeck.de

Zusammenfassung. In diesem Papier wird ein neuartiger Ansatz für Test und Fehlersuche in komplexen Autonomen Systemen vorgestellt. Basis der Lösung ist die Analyse von Trace-Daten, die von dem Zielsystem über eine dedizierte, häufig schon vorhandene Prozessoreinheit zur Verfügung gestellt werden. Im Vergleich mit dem aktuellen Stand der Technik wird der Ansatz eingeordnet, und die besonderen Vorzüge hinsichtlich der Nicht-Intrusivität und der unbegrenzten Überwachungslaufzeit herausgestellt.
Nach einem kurzen Einblick in die Funktionsweise des Systems folgen zwei Anwendungsbeispiele, die Code-Coverage-Messung und die dynamischen Analyse, mit welchen der praktische Nutzen des Systems verdeutlicht und ein Einblick in die Handhabung der neuen Methode geliefert wird. Die Möglichkeiten des Systems reichen von einfachen Zeitmessungen, über Wirkkettenmessungen (WCRT Abschätzung), bis hin zu komplexen funktionalen Tests. Wesentlicher Bestandteil für die dynamische Analyse stellt dabei die Verifikationssprache TeSSLa dar, welche dem Leser anhand von kurzen, prägnanten Beispielen näher gebracht wird. Abschließend werden Anforderungen an das zu beobachtende System gegeben, um den Einsatz des vorgestellten Werkzeugs zu ermöglichen.

1 Allgemeine Einführung

Autonome Systeme profitieren außerordentlich von immer leistungsfähigeren Prozessoren bzw. werden durch diese erst möglich. Die Komplexität der Systeme steigt, der Umfang der Software nimmt zu, und es werden immer mehr auch sicherheitskritische Anwendungsgebiete erschlossen.

Da die relative Fehlerwahrscheinlichkeit von Software mit ihrer Komplexität [1] sowie auch mit der Verwendung nichtdeterministischer Strukturen in Form

* This work was funded in part by the EU H2020 project 732016 *COEMS* and by the BMBF project *ARAMiS II* with funding ID 01 IS 16025.

von Multi-Core-Prozessoren steigt, sind Entwickler und Anwender mit einer zunehmenden Zahl von Fehlern, die ungünstigerweise auch noch im Release-Code verbleiben können, konfrontiert.

Die Bandbreite dieser Fehler reicht von Fehlern in der Anforderungsspezifikation, trivialen Implementierungsfehlern bis hin zu komplexen nicht-deterministischen Fehlerbildern. Aufgrund fehlender oder zu stark vereinfachter Modelle stößt auch die statische Analyse an ihre Grenzen, so dass die dynamische Analyse autonomer Systeme zunehmend an Bedeutung gewinnt.

Die Standards für die Entwicklung von sicherheitskritischen Systemen [2–7] definieren konkrete Anforderungen an den Testprozess, die anzuwendenden Testtechniken und den Nachweis der Vollständigkeit dieser Tests (strukturelle Überdeckung). Die Normen orientieren sich an dem V-Modell und unterscheiden verschiedene Testebenen (Modultest, Integrationstest, Systemtest). Die Weiterentwicklung der Normen lässt erkennen, dass zunehmend gefordert wird, Tests konsequent gegen die Anforderungsspezifikation (*requirements-based tests*) durchzuführen[1]. Dieser Nachweis kann im Wesentlichen nur mit **T**est **H**öherer **E**benen (THEs) vollzogen werden.

Als Grundlage für Zertifizierungen von Software und software-gesteuerten eingebetteten Systemen, speziell im sicherheitskritischen Bereich, muss die vollständige strukturelle Überdeckung des Quellcodes durch Tests nachgewiesen werden. Je nach Kritikalität der Anwendung ist der Nachweis zu erbringen, dass während der Tests alle Instruktionen (Statement Coverage, C0), alle Sprünge (Branch Coverage, C1) oder alle relevanten Bedingungskombinationen (wie beispielsweise die modifizierte Bedingungs-/Entscheidungsüberdeckung, MC/DC) verwendet wurden. Dabei lassen es die Normen weitgehend offen, auf welcher Testebene die Messung der strukturellen Überdeckung erfolgen soll.

Gerade in autonomen Systemen werden diese Tests immer wichtiger, um das Zusammenspiel komplexer Komponenten – die oftmals von verschiedenen Herstellern kommen und nicht im Source-Code vorliegen – verifizieren zu können. Die für die höheren Testebenen erforderliche dynamische Analyse verspricht folgende Vorteile:

- Strukturelle Tests auf höheren Testebenen,
- Erkennung von Zeitschrankenverletzungen und
- Effiziente Analyse von Fehlverhalten im Feld.

Bisher werden dynamische Analysen nicht umfassend eingesetzt, da komplexe Systeme nur mittels Software-Instrumentierung oder mittels Offline-Analyse von Trace-Daten beobachtbar sind. Damit lässt sich praktikabel keine umfassende funktionale Testabdeckung erreichen, welche aber durch die stetig wachsende Anzahl von - insbesondere sporadischen - Fehlerbildern notwendig ist.

[1] "DO-178C was intentionally strengthened over its predecessor DO-178B to ensure acceptable requirements via 178C's mandate to trace structural coverage analysis to requirements-based tests (RBT)." [8]

2 Stand der Technik

Bei der Software-Instrumentierung protokolliert hinzugefügter Code die Ausführung des Programms während eines Testlaufs oder während der Fehlersuche. Die Erweiterung des Programmcodes erfolgt dabei automatisiert unter der Verwendung klassischer Werkzeuge wie VectorCast [9], Tessy [10] oder gcov [11]. Dieses Verfahren hat den Nachteil, dass der instrumentierte Programmcode zusätzlichen Speicherplatz sowie zusätzliche Rechenzeit benötigt, was sich besonders bei höheren Testebenen (Integrations- und Systemtest) und nichtdeterministischen Systemen nachteilig auswirkt. Weichen hier bei der Beobachtung des Zusammenspiels von Software- und Hardwarekomponenten Ausführungszeiten und Speicherlayout von der finalen Lösung ab, sind die gewonnenen Ergebnisse nur bedingt verwertbar. Zudem kommt bei sicherheitskritischen Systemen die Einschränkung hinzu, dass bereits zertifizierter Code nicht ohne weiteres modifiziert werden darf. Dies führt zu einem erheblichen Mehraufwand, speziell bei der Fehlersuche im Feld.

Für Modultests ist die software-basierte Instrumentierung ein geeignetes Verfahren, da hier Ausführungszeit und Speicherbedarf nur eine untergeordnete Rolle spielen. Anders ist die Situation bei THEs, in denen auch das Zusammenspiel von Software- und Hardwarekomponenten überprüft wird. Weichen hier Ausführungszeiten und Speicherlayout von der finalen Lösung ab, können Aussagekraft und Verwertbarkeit der gewonnenen Ergebnisse eingeschränkt sein.

Eine andere technische Möglichkeit zur Messung der strukturellen Überdeckung besteht in der Auswertung von Trace-Daten, mit denen der verwendete Prozessor Informationen über den abgearbeiteten Programmfluss liefert [12]. Diese Informationen werden mittels einer im Prozessor integrierten Hardware-Einheit ausgegeben, ohne dass der Programmablauf beeinflusst wird. Die sehr breitbandigen Trace-Daten (einige Gbit/s) werden in einem Speicher abgelegt. Nach dem Ende des Testlaufs wird auf einem PC der Programmfluss rekonstruiert und die strukturelle Überdeckung ermittelt. Die notwendige Zwischen-speicherung und die dadurch begrenzte Aufzeichnungsdauer begründet die Limitierung dieses Verfahrens: selbst bei einer Speichergröße von mehreren GByte sind nur einige Sekunden des Programmablaufs rekonstruierbar. Somit ist auch die Offline-Trace-Analyse für dynamische Tests auf höheren Ebenen und die Fehlersuche in komplexen autonomen Systemen für viele Anwendungsfälle eher ungeeignet.

3 Online-Trace-Analyse

Das Arsenal an Werkzeugen zur dynamischen Analyse komplexer Systeme wurde nun um ein weiteres Tool erweitert, indem der beschriebene Ansatz der Offline-Trace-Analyse zu einer echtzeitfähigen Lösung ohne Zwischenspeicherung weiterentwickelt [13–17, 26] wurde. Dieses System ist erstmals in der Lage, die eingehenden Trace-Daten kontinuierlich zu verarbeiten und den Kontrollfluss in Echtzeit zu rekonstruieren. Dafür werden die von den Trace-Einheiten des Prozessors ausgegebenen Trace-Daten in einem FPGA online verarbeitet.

Beobachtungsziel	**Nachricht**	**Architektur**		
		x86 [18] (Intel PT)	ARM [19][2]	NXP QorIQ PPC [20, 21]
Kontrollfluss	Synchronisierungsnachricht (Aktuelle Instruktionsadresse)	FUP	I-Sync-Message	Indirect Branch History Message
	ausgeführter direkter Sprung, nicht ausgeführter direkter oder indirekter Sprung	TNT	Atom-Message	
	Ausgeführter indirekter Sprung, Exception	TIP	Ibranch-Message	
Daten	Daten-Zugriff	n.a.	n.a.[3]	Data Trace Message
	Hardwareunterstützte Instrumentierung	PTWrite	ITM, STM[4]	Data Acquisition Message
Betriebssystem	Anzeige eines Taskwechsels (abhängig von OS-Unterstützung)	PIP	Ownership-Message	Ownership Trace Message
Peripherie-einheiten	anwenderspezifische Trace-Nachrichten (umfassende Beobachtung komplexer Peripherieeinheiten	n.a.	zum Teil STM	Bus Trace, DDR Trace, Data Path Trace

Tabelle 1. Architekturspezifische Inhalte von Trace-Nachrichten

Obwohl es je nach Prozessor-Architektur Unterschiede im Trace-Protokoll gibt, werden im Wesentlichen identische Informationen übertragen:

Die Trace-Daten werden via einer seriellen Hochgeschwindigkeitsschnittstelle mit einer Bandbreite von einigen Gbps ausgegeben. Es werden entweder dedizierte Schnittstellen (basierend auf dem Aurora-Protokoll [22]) oder Systemschnittstellen (PCIe, USB) verwendet, wobei letztere durch die Nutzung prozessor-interner Ressourcen zu einer Beeinflussung des Programmlaufs führen kann. Zudem wird bei älteren oder einfacheren Prozessoren noch eine parallele Trace-Schnittstelle verwendet, die aber viele I/O-Pins benötigt sowie störanfällig ist, da eine Sicherungsschicht für die Bitübertragung oftmals nicht implementiert ist. Eine Übersicht über die Inhalte von Trace-Nachrichten bei verschiedenen Architekturen ist in Tabelle 1 gegeben.

Die technische Herausforderung besteht nun darin, den Trace-Datenstrom vorzuverarbeiten und den Kontrollfluss der CPU(s) zu rekonstruieren. Hier kommt verkomplizierend hinzu, dass der Adressversatz ausgeführter direkter Sprünge nicht Bestandteil der Trace-Nachrichten ist, sondern dass nur die Information übertragen wird, ob ein direkter Sprung ausgeführt wurde oder nicht. Mittels vorberechneten, in einem externen Speicher hinterlegten Lookup-Tabellen kann

[2] Hier wird Bezug auf die PFT Einheiten von ARM CoreSight genommen. Die konkreten Bezeichnungen können bei ETM abweichen.

[3] Im Gegensatz zu PFT unterstützt ETM auch Datentrace. Allerdings ist die Verfügbarkeit der Datentrace-Module bei aktuellen Prozessoren eingeschränkt.

[4] ITM, STM sind separate Einheiten, die von mehreren CPUs geteilt werden.

durch massive Parallelisierung der Trace-Verarbeitung und der Speicherzugriffe auch der Kontrollfluss für CPUs berechnet werden, die schneller als 1 GHz getaktet sind. Das Verfahren ist sowohl für Bare-Metal- als auch für Multitasking-Betriebssysteme anwendbar.

Die zugrunde liegende Hardware besteht aus einem komplexen FPGA (Xilinx Virtex-7 bzw. Xilinx Zynq Ultrascale+) mit mehreren RLDRAM3-Speicher-Bänken [23]. Dieser Speichertyp stellt einen Kompromiss zwischen SRAM (kein Overhead bei wahlfreiem Zugriff, geringe Speicherkapazität, teuer) und DDR4 SDRAM (großer Overhead bei wahlfreiem Zugriff, hohe Speicherkapazität, günstiger Preis) dar.

4 Anwendungsgebiet: Code Coverage

Ein Anwendungsgebiet der kontinuierlichen Rekonstruktion des Kontrollflusses ist die Messung der strukturellen Testabdeckung (Code Coverage) bei der Durchführung von THEs. Üblicherweise erfolgt diese auf der Ebene der Modultests. Dazu wird der zu untersuchende Programmcode instrumentiert und ein funktionaler sowie ein struktureller Test durchgeführt. Danach wird die Instrumentierung entfernt und der funktionale Test erneut durchgeführt, um sicherzustellen, dass die Entfernung der Instrumentierung zu keiner Änderung der Funktionalität geführt hat. Für Modultests ist diese Vorgehensweise erprobt und zuverlässig, allerdings müssen bei Verwendung der Instrumentierung auf höheren Testebenen (Integrationstest, Modultest) einige Einschränkungen berücksichtigt werden. Diese resultieren aus dem veränderten Verhalten der zu testenden Anwendung - der durch die Instrumentierung eingebrachte Code benötigt zusätzlichen Speicherplatz und Rechenzeit. In optimierten eingebetteten Systemen steht dieser zusätzliche Speicherplatz nicht immer zur Verfügung, zudem kommt es zu einer Änderung des Laufzeitverhaltens. Sollen nun Laufzeiteigenschaften getestet werden, so wird diese Untersuchung durch die Instrumentierung und damit die Aussagekraft des Tests beeinflusst.

Die Messung der strukturellen Abdeckung auf Object-Code-Ebene liefert andere Information als bei einer Messung auf Quellcode-Ebene und wird ausführlich in [24] diskutiert. Durch Auswertung der vom Compiler generierten Debug-Informationen kann die gemessene Object-Code-Coverage auf die von vielen Tools unterstützte Source-Code-Coverage gemappt werden.

Bei der üblichen Verwendung von Compiler-Optimierungen unterstützen die Debug-Informationen (DWARF, ELF) dieses Mapping oftmals nur unzureichend — Aktivitäten für eine Verbesserung in diesem Bereich sind bereits gestartet.

Wesentliche Vorteile von strukturellen THEs sind:

- Die Messung der strukturellen Überdeckung während der Ausführung von THEs stellt eine mathematisch exakte Methode dar, um an dieser Stelle Fehler und Unvollständigkeiten zu erkennen. Die Methode leistet damit einen wichtigen Beitrag zur Verbesserung der Softwarequalität.
- Der Aufwand für die Entwicklung struktureller Tests auf Modultestebene kann deutlich reduziert werden, wenn hier die mittels struktureller THE bereits erzielte Testabdeckung berücksichtigt wird.

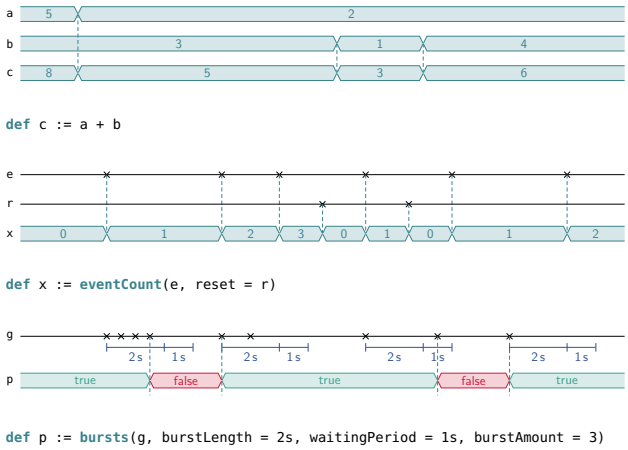

Abb. 1. Ein Beispiel für verschiedene TeSSLa-Spezifikationen.

5 Anwendungsgebiet: Dynamische Analyse

Bei der kontinuierlichen Rekonstruktion des Kontrollflusses ist es möglich, bestimmte Instruktionsadressen zu markieren. Bei der Ausführung dieser Instruktionen werden Elemente in einen Ereignisstrom eingefügt, welcher dann im FPGA auf bestimmte Eigenschaften untersucht werden kann. Diese Ereignisverarbeitungseinheiten sind hochsprachlich [25] konfigurierbar und können eine Vielzahl von temporalen Eigenschaften parallelisiert überwachen. Dabei ist im FPGA nur eine Parametrisierung der Ereignisverarbeitungseinheiten erforderlich, eine aufwändige anwendungsspezifische Synthese von Logikstrukturen ist nicht nötig. Somit kann eine Änderung der hochsprachlichen Eigenschafts-Beschreibung binnen Sekunden auf einen Trace-Datenstrom angewandt werden.

Die von uns genutzte Sprache zur Beschreibung der vorher genannten temporalen Eigenschaften ist die Temporal Stream-based Specification Language (TeSSLa, [25]). TeSSLa beschreibt Eigenschaften in Form von Variablen und Gleichungssystemen, denen eine Menge von Eingabeströmen gegeben wird und die durch Stromtransformationen eine Menge von Zwischenströmen und, am Ende, Ausgabeströmen erzeugt. Ein Strom ist hierbei eine Folge von Events, die jeweils aus Wert und Zeitstempel bestehen. Hierbei können Eingabeströme aus beliebigen Quellen stammen, wie aus einer Datei eingelesen werden, aber auch dynamisch an Funktionen oder Variablen in Programmen gekoppelt werden, so dass diese immer ein Event enthalten, wenn z.B. die Funktion aufgerufen oder die Variable belegt wird.

Ein Beispiel für TeSSLa-Spezifikationen findet sich in Abbildung 1. Die erste Spezifikation zeigt das Aufaddieren von zwei Strömen und das Erzeugen eines neuen Stroms mit dem Ergebnis. Das zweite Beispiel zeigt das Zählen von Events mittels der eventCount Funktion. Die dritte zeigt die Spezifikation eines kom-

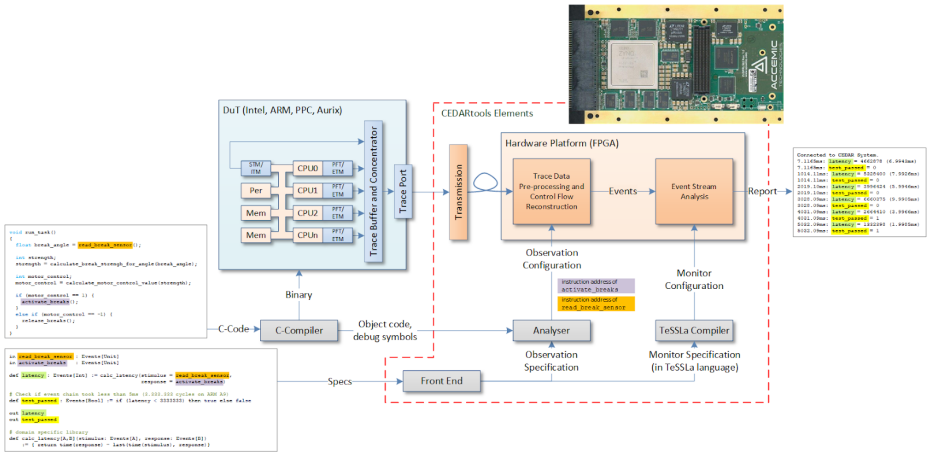

Abb. 2. System zur kontinuierlichen dynamischen Analyse autonomer Systeme

plexen Burst-Patterns. Dank des starken Makro-Systems hinter TeSSLa können solch komplexe Spezifikationen durch die Verwendung bestehender Operationen leicht realisiert werden. Zu Grunde liegen dem sechs Operatoren, zu denen unter anderem **time**(x) gehört, welcher einen Strom mit den Zeitstempeln jedes Events des Eingabestroms zurückgibt und **last**(x, y), welcher, immer wenn ein Event auf dem Strom y auftritt, ein Event mit dem Wert des letztens Events des Stroms x zurückgibt und daher das Zugreifen auf vergangene Werte ermöglicht.

Das in Abbildung 2 angegebene, stark vereinfachte Beispiel zeigt die Messung der Latenz eines Bremssystems. Die Funktion run_task() wird periodisch auf dem DuT (Device under Test) ausgeführt. Der Task liest bei jeder Ausführung den Winkel des Bremspedals - über die Funktion read_break_sensor() - ab und kontrolliert anhand dessen die Bremsen des Fahrzeugs. In dem vereinfachten Beispiel geschieht dies, in dem bei einer starken Bremsung die Funktion activate_breaks() aufgerufen wird. Wird die Bremse gelöst, geschieht das wiederum durch die Funktion release_breaks().

Die Ausführung des Tasks kann auf dem Zielsystem - insbesondere auf Multi-Kern-Systemen - unterschiedlich viel Zeit benötigen, und mitunter durch andere Tasks, Interrupts, Zugriffe auf gemeinsame Ressourcen wie Speicher, oder andere Einflüsse unterbrochen werden. Häufig steht für solche End-zu-End Reaktionszeiten aber nur ein begrenztes Zeitbudget zur Verfügung, insbesondere in sicherheitskritischen Abläufen.

Mit CEDAR lässt sich nun das Einhalten dieses Zeitbudgets sicherstellen. Die TeSSLa-Spezifikation in Abbildung 3 ermittelt die Latenz vom Lesen des Sensors bis zum Aktivieren der Bremsen. Zudem wird eine Grenze von 5 Millisekunden festgelegt, die beim Überschreiten einen fehlgeschlagenen Test signalisiert.

Als Eingänge fungieren die Aufrufe der Funktionen read_sensor() und activate_breaks(), welche über die Benennung der Eingangsströme entsprechend definiert werden. Neben Funktionsaufrufen können auch beliebige Stellen im

```
in read_sensor : Events[Unit]
in activate_breaks : Events[Unit]

def latency : Events[Int] := time(activate_breaks) - last(time(read_sensor), activate_breaks)

# Check if event chain took less than 5ms (3.333.333 cycles on ARM A9)
def test_passed : Events[Bool] := (latency < 3333333)

out latency
out test_passed
```

Abb. 3. TeSSLa-Spezifikation für vereinfachtes Beispiel eines Bremssystems

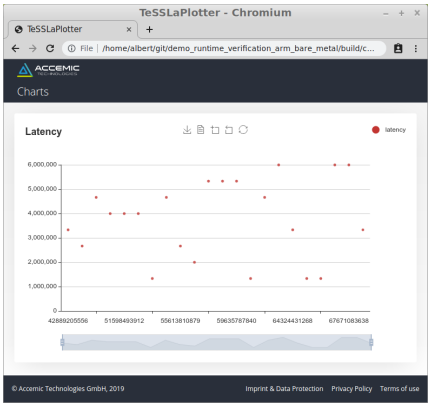

Abb. 4. Graphische Ausgabe der Messwerte

Quellcode, Task-Wechsel oder Datenzugriffe als Eingänge für die Spezifikation verwendet werden. Das Schreiben der Spezifikation an sich wird durch eine umfassende Bibliothek vereinfacht, die gängige Muster durch Funktionen bereits implementiert und bereitstellt.

Ohne Modifikation der Applikation kann nun über einen beliebig langen Zeitraum bestimmt werden, ob die Reaktionszeit innerhalb einer vorgegebenen Schranke geblieben ist.

Die Ausgabe der Daten erfolgt neben einer Textausgabe, auch in graphischer Form mit verschiedenen Diagrammen, wodurch Normabweichungen des Verhaltens in der Anwendung einfach erfassbar werden (siehe Abbildung 4).

Durch die Beobachtung auf Object-Code-Ebene ergibt sich der Vorteil der Unabhängigkeit von Compiler und Programmiersprache, lediglich beim Mapping von hochsprachlichen Elementen (z.B. Funktionsaufruf, Exceptions, Speicherzugriffe) auf die Object-Code-Ebene bzw. umgekehrt (Object-Code-Coverage auf Source-Code-Coverage) werden die vom Compiler ausgegebenen Debug-Informationen (z.B. DWARF, IEEE-595) verwendet.

Dieses Analyseverfahren kann sowohl bei Integrationstests, bei Systemtests als auch bei der Fehlersuche im Feld eingesetzt werden. Im Falle eines Fehlers, können umfangreiche Logdaten analysiert werden, um dessen Ursache zu lokalisieren. Diese Daten beinhalten unter anderem eine Historie der zurückliegenden Events, die für die Spezifikation verwendet wurden, Betriebssystem-Events (z. B.

Taskwechsel, Interrupts, usw.) und auch den Programmfluss vor dem Auftreten des Fehlers.

Weit über das gezeigte Beispiel hinaus lässt sich CEDAR nun für umfangreiche Timing-Analysen verwenden. In realen Systemen bestehen Wirkketten meist aus mehreren zehn bis hundert Gliedern. CEDAR bietet die Möglichkeit für solche Wirkketten eine Abschätzung der WCRT (Worst Case Response Time) zu liefern, welche durch ausgiebige Tests des Systems fundiert wird. Auch eine Abschätzung der WCET (Worst Case Execution Time) von Funktionen ist mit dem CEDAR System ermittelbar, in dem die Funktionslaufzeiten ohne jegliche Unterbrechungen durch Interrupts o.ä. gemessen werden. Neben der Timing-Analysen können auch Kennzahlen der strukturellen Abdeckung der Software gemessen werden. Sowohl Code-Coverage als auch Data-Coverage Statistiken können genutzt werden, um die Qualität der Tests beurteilen zu können. Dabei ermöglicht der nicht-intrusive Ansatz von CEDAR nicht nur den Einsatz auf Unit-Test Ebene, sondern erlaubt auch die Messung in Tests auf System-Ebene, welche vorher durch andere Lösungen nur mit erheblicher Veränderung des Laufzeitverhaltens der zu beobachtenden Anwendung denkbar gewesen wäre.

Ein verantwortungsvolle Projektplanung wird – schon im Hinblick auf die die sicher zu erwartenden Post-Release-Defects – Vorkehrungen beinhalten, um eine umfassende Beobachtbarkeit zu gewährleisten. Dies bedeutet:

- Die Trace-Schnittstelle muss verfügbar sein. Für Intel-Prozessoren sollte der entsprechende USB-Port verfügbar sein, für andere Architekturen muss im Hardwaredesign der Zugang zu den entsprechenden Trace-Schnittstellen (meist Aurora oder parallel) berücksichtigt werden. Kurzsichtige Kosteneinsparungen an dieser Stelle können den Projekterfolg gefährden.
- Die Initialisierung der Trace-Schnittstelle muss sichergestellt sein. Entweder geschieht dies während der Startup-Routine (fest implementiert oder im BIOS einstellbar) oder durch mögliche externe Zugriff auf die entsprechenden Control-Register (z.B. via JTAG). Bei beiden Varianten sind die Auswirkungen auf die Safety- und/oder Security-Architektur zu berücksichtigen.
- Da viele Prozessor-Architekturen nur über einen eingeschränkten Datentrace verfügen, sollten zukünftige Anforderungen an die Beobachtbarkeit (speziell auch für die Fehlersuche im Feld) analysiert werden. Daraus resultiert dann z.B. die Verwendung hardwareunterstützter Instrumentierung (PTWrite, STM, Data Acquisition Messages) oder das Mappen relevanter Variablen in Speicherbereiche, die durch den Datentrace beobachtet werden können.

Das im Vortrag vorgestellte dynamische Analyseverfahrens übernimmt die Beobachtung und kontinuierliche Überwachung von komplexen autonomen Systemen auf der Basis der kontinuierlichen Auswertung von Prozessor-Trace-Daten.

Die kontinuierliche Beobachtung garantiert die Erkennung auch selten auftretender Fehlfunktionen in Langzeittests oder im Feld. Sie ermöglicht damit die Durchführung effektiver Tests, eine frühzeitige Fehleridentifizierung und eine zielgerichtete Fehlerbehebung. Damit wird die Qualität, die Zuverlässigkeit und die Ausfallsicherheit autonomer Systeme erhöht sowie das Risiko und die Schwere von Post-Release-Defekten reduziert.

Literaturverzeichnis

1. C. Jones und O. Bonsignour, The Economics of Software Quality. Addison-Wesley, 2011.
2. 'ISO 26262:2018. Road vehicles – Functional safety'. International Organization for Standardization Std., 2018.
3. 'ED-12C - Software Considerations in Airborne Systems and Equipment Certification'. EUROCAE, 2011.
4. 'ISO 25119:2010 Tractors and machinery for agriculture and forestry – Safety related parts of control systems'. 2010.
5. 'IEC 61508:2010 Functional safety of electrical/ electronic/programmable electronic safety-related systems'. International Electrotechnical Commission, 2010.
6. 'DIN EN 50657:2017 Bahnanwendungen – Anwendungen für Schienenfahrzeuge – Software auf Schienenfahrzeugen'. 2017.
7. 'EN 50128:2011 Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems'. European Commitee for Electrotechnical Standardization, 2011.
8. V. Hilderman, 'DO-178C Best Practices For Engineers & Managers'. AFusion / Jama Software.
9. VectorCAST embedded software testing platform. Vector Informatik GmbH.
10. TESSY - Test System. Razorcat Development GmbH.
11. gcov test coverage program. gnu.org.
12. ARM IHI 0035B - CoreSightTM Program Flow Trace TM PFTv1.0 and PFTv1.1 Architecture Specification. ARM Limited, 2011.
13. B. Dreyer, C. Hochberger, S. Wegener, und A. Weiss, „Precise Continuous Non-Intrusive Measurement-Based Execution Time Estimation“, in 15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015), Dagstuhl, Germany, 2015, Bd. 47, S. 45–54.
14. A. Weiss und A. Lange, „Trace-Data Processing and Profiling Device“, US 9286186B2, 15-März-2016.
15. A. Weiss und A. Lange, „Trace-Data Processing and Profiling Device“, EP2873983A1, 20-Mai-2015.
16. B. Dreyer, C. Hochberger, A. Lange, S. Wegener, und A. Weiss, „Continuous Non-Intrusive Hybrid WCET Estimation Using Waypoint Graphs“, in 16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016), Dagstuhl, Germany, 2016, Bd. 55, S. 1–11.
17. N. Decker u. a., „Online Analysis of Debug Trace Data for Embedded Systems“, gehalten auf der Design, Automation and Test in Europe, Conference and Exhibition (DATE 2018), Dresden, Germany, 2018.
18. Intel® 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation, 2016.
19. CoreSightTM Architecture Specification v2.0 ARM IHI 0029B. ARM Limited, 2013.
20. IEEE-ISTO, 'The Nexus 5001 Forum - Standard for a Global Embedded Processor Debug Interface', IEEE-ISTO 5001TM-2012, Jun. 2012.
21. e500mc Core Reference Manual. Freescale Semiconductor, Inc., 2012.
22. Aurora Protocol Specification. Xilinx Inc., 2007.
23. '1.125Gb: x18, x36 RLDRAM 3'. Micron Technology, Inc., 2016.
24. 'Position Paper CAST-17: Structural Coverage of Object Code'. FAA Certification Authorities Software Team (CAST), Jun-2003.
25. L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, und D. Thoma, „TeSSLa: Temporal Stream-Based Specification Language“, in Formal Methods: Foundations and Applications, Cham, 2018, S. 144–162.
26. T. Scheffel u. a., „Rapidly Adjustable Non-Intrusive Online Monitoring for Multicore Systems“, gehalten auf der Brazilian Symposium on Formal Methods (SBMF), 2017.