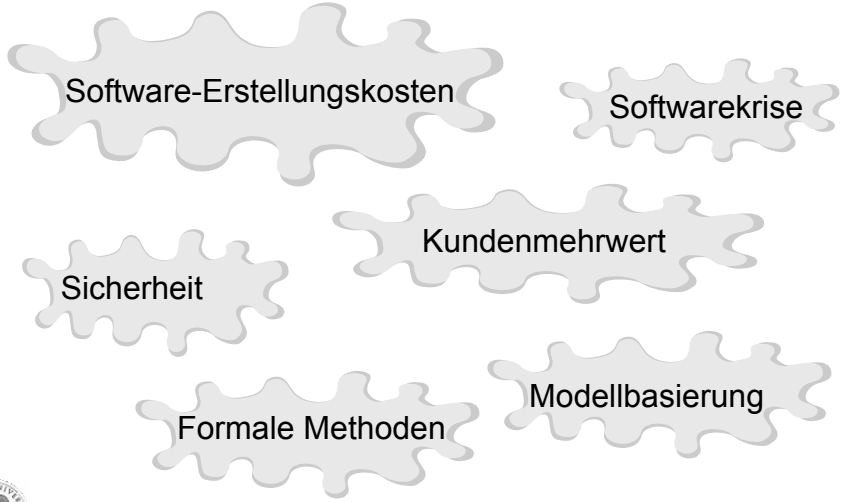
Modellbasiertes Testen von Software

Prof. Dr. Holger Schlingloff

Humboldt-Universität zu Berlin,
Institut für Informatik

Fraunhofer Institut für Rechnerarchitektur
und Softwaretechnik FIRST

Motivation



Begriffseingrenzung

Experimentieren = Ausführen *einzelner* Versuche zur Erlangung einer neuen Erkenntnis

Probieren = experimentelles Feststellen der Qualität eines Objekts

Testen = *systematisches* Probieren nach verschiedenen Qualitätskriterien

Prüfen = Testen einer *Serie* gleichartiger Objekte

Abgrenzung

Validation:
Ist es die richtige Software?

Verifikation:
Die Software ist richtig!

Test:
Ist die Software richtig?

Debugging:
Warum ist die Software nicht richtig?

Testen versus Verifizieren

- Testen kann nur die Anwesenheit von Fehlern zeigen, nicht deren Abwesenheit

  --- ABER ---

- Verifikation großer Systeme ist komplex
- Source-Code ist manchmal nicht verfügbar
- Fehler auf vielen Ebenen möglich (SW/OS/HW)

Gates' Mathematik

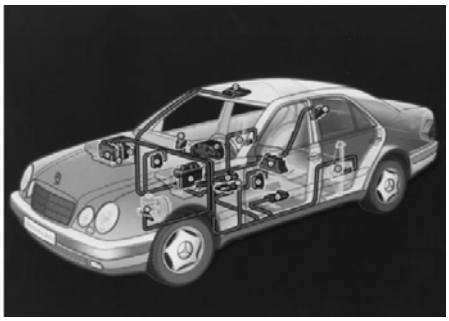
Genie oder Unfähigkeit

Redmond (ag) – Microsofts Java-Compiler widersetzt sich nicht nur Suns Kompatibilitätstest, er definiert auch die Fakultät etwa von 5 neu.

Bill Gates ist ein Freund von kleinen Zahlen. Er hat nicht nur erfolgreich die Anzahl verbreiteter Betriebssysteme reduziert, er schickt sich nun auch an, das mathematische Modell der Fakultät zu revolutionieren. Tüftler haben entdeckt, daß ihre Java-Programme nach der Optimierung mit Microsofts Just-in-time-Compiler neue Ergebnisse liefern – die Zahlen werden kleiner. Die Fakultät von 5 ist dadurch auf überschaubare 15 geschrumpft, vor Gates war sie noch dreistellig.

anderes Beispiel: Automobil

- 40-80 eingebettete Steuergeräte
- 80 K Lines of Code für einen Airbag
- Rückrufaktionen mit Millionenkosten



Vorteile von SW im Automobil

- Kosten:
  - hohe Stückzahlen
  - höhere Entwicklungskosten, geringere Fertigungskosten
- Flexibilität:
  - Änderungs- und Upgrademöglichkeiten in Entwicklung und Betrieb
  - verbesserte Diagnose
- Innovation:
  - Gewichtsreduktion direkt und durch verringerten Kabelaufwand
  - Zuverlässigkeit da keine Alterung
- Mehrwert:
  - Realisierung der meisten neuen Alleinstellungsmerkmale
  - hohe Kundenwahrnehmung

Probleme

- 127 Rückrufaktionen in 2002 in Deutschland, davon ca. 15% Softwarefehler (Tendenz steigend)
- Kosten jeweils in Millionenhöhe (100€/Fahrzeug fix)

  Beispiele:
  - Türöffnung entriegelt nicht mehr
  - Kurzschluss in der Innenbeleuchtung führt zu totalem Batterieausfall
  - Aufprallsensor für Seitenairbag löst nicht aus
- Zulassungsproblematik
  - z.B. X-by-wire oder elektronische Abstandshaltung

Testziel

- Qualität = Übereinstimmung mit den Anforderungen

z.B.: Funktionalität, Zweckdienlichkeit, Robustheit, Zuverlässigkeit, Sicherheit, Effizienz, Benutzbarkeit, Geschwindigkeit, ...

➔ verschiedene Testmethoden

➔ Formulierung von überprüfbaren Anforderungen

Notwendigkeit der Formalisierung

- Natürliche Sprache mehrdeutig!
- Beispiel:

> „alle 30 Sekunden sollen die Werte der Sensoren abgelesen werden; wenn die Standardabweichung 0,25 überschreitet, soll die Normalisierungsprozedur ausgeführt werden, anschließend sollen die Werte an das Analysepaket weitergegeben werden."

Akzeptanztest ergibt falsche Analysewerte.
Problem: Komma!

Formale Anforderungsdefinition

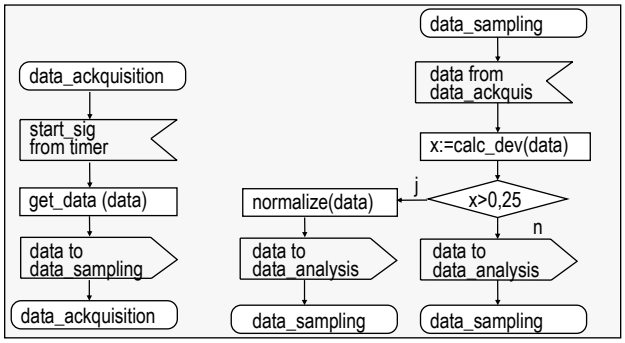
(a) Festlegung der Schnittstellen und Ereignisse
(b) Festlegung des reaktiven Verhaltens

(a)

Methoden:
get_data (...)
calc_dev(...)
normalize (...)
set_timer (...)

Signale:
data: ...
deviation: ...
start_sig: ...

(b)



Deklarative statt operationaler Beschreibungsformen
(*was* statt *wie*)

Alle Werte sollen normalisiert analysiert werden.

Logische Spezifikation der gewünschten Eigenschaften
(formale Grundlage)

For all x exists y: y=normalize(x) and sometime analyzed(y)

→ Spezifikationsbasierte Tests

Whitebox- und Blackbox-Test

- Codebasierte Tests orientieren sich an der Struktur des zu testenden Programms
  - Stümpfe und Treiber
  - Äquivalenzklassenbildung, Grenzwertanalyse, Entscheidungstabellen
  - JUnit, Cantata, Tessy, ...
    (> 50 Tools in www.testingfaqs.org)
- Spezifikationsbasierte Test orientieren sich an der Beschreibung der Programmeigenschaften
  - Capture-Replay: WinRunner, Rational Robot, ...
  - Modellbasiert: separate Beschreibung

Definition: Modell

Brockhaus: Modell (von lat. modulus)

1. **Vorbild:** Aufbau oder Form nach der das eigentliche Werk geschaffen wird; gedanklicher Entwurf
2. **Abbild:** vereinfachende bildliche oder mathematische Darstellung von Strukturen, Funktionsweisen oder Verlaufsformen; Wiedergabe eines Gegenstandes in verkleinertem Maßstab zu Studien- und Versuchszwecken

Modelle beim Testen

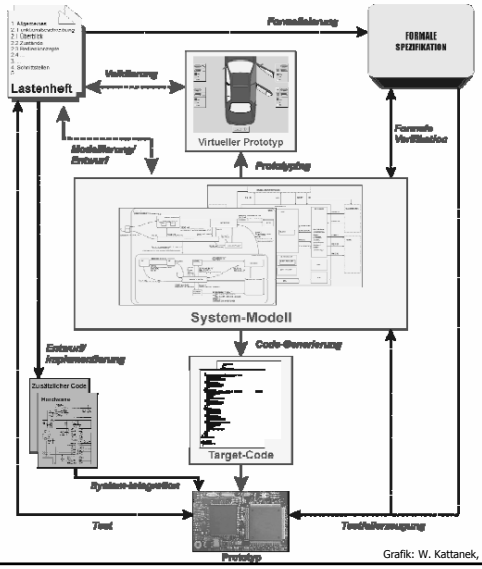
- Systemmodell

  Beschreibung des gewünschten Verhaltens des Systems in der modellierten Umgebung

- Umgebungsmodell

  Beschreibung der (physikalischen oder logischen) Umgebung des zu implementierenden Systems

modellbasierter Entwurf



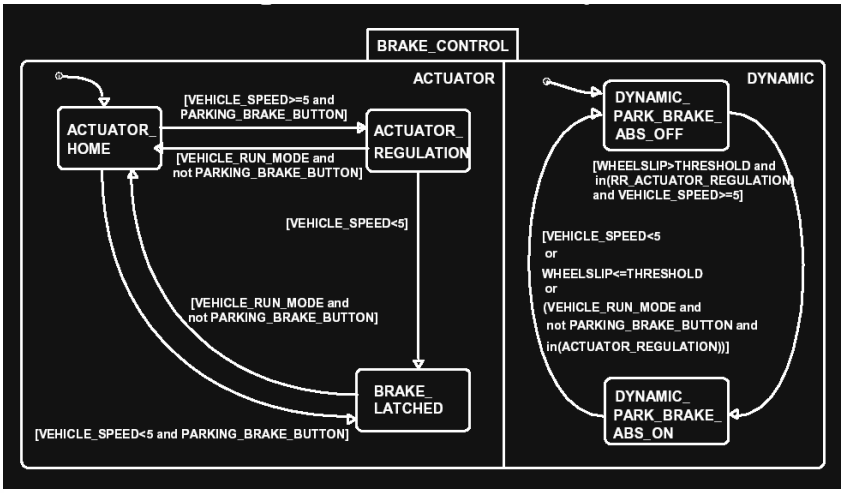
- durchgängiger, kompatibeler Entwurfsprozess
- ingenieursgerechte Notation und Methodik
- Werkzeugunterstützung

Grafik: W. Kattanek, astec

StateCharts

- von D. Harel 1987 eingeführte Erweiterung von endlichen Automaten
- Strukturierung: Hierarchie und Parallelität
- Variablen, Kommunikation per Broadcast
- als Statechart Diagrams in UML eingeflossen
- Semantikprobleme
- vor allem im Automobilbereich viel verwendet
- iLogix-Tools: StateMate, Rhapsody

Beispiel: Bremskontrolle



Verwendung von StateCharts

- Modellierung kontinuierlicher Komponenten mit MatrixX oder Matlab/Simulink
- Möglichkeit der Animation der Zustandsübergänge
- Codegenerierung
- Testgenerierung

Anderer Formalismus: CSP

- Communicating Sequential Processes, Hoare 85

Syntax:

$$\alpha ::= STOP \mid SKIP \mid q \mid (a \longrightarrow \alpha) \mid (\alpha \,\Box\, \alpha) \mid (\alpha \sqcap \alpha) \mid (\alpha; \alpha) \mid (\alpha \underset{A}{\parallel} \alpha) \mid (\alpha \overset{t}{\triangleright} \alpha) \mid \nu q\, \alpha \mid \alpha\{q_1 := q_2\}$$

- Synchronisation, Komposition, Rekursion, Timeout
- operationelle Semantik: Zustandsübergangssystem
- Realisierung in FDR2:
  Berechnung von Verfeinerungen zwischen Prozessen

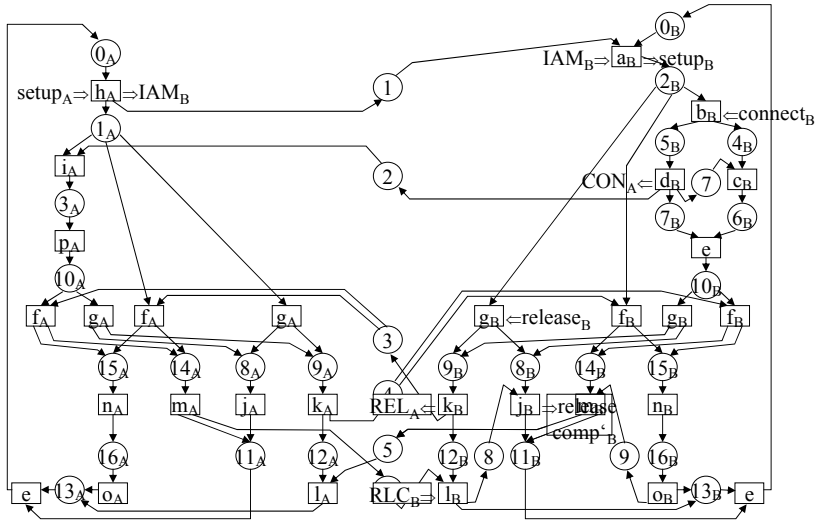
Beispiel: Telekommandos

At any given moment, it is possible to send telecommands for turning any device on or off. All switching operations neccessary to activate or deactivate this device must be performed within a given time constant.

```
SPEC = ( SWITCHDEV [|{| Tau_nextTC |}|] TCTIM ) ||| TIMCHK
SWITCHDEV = Tau_nextTC -> (
(Com_PYRO_PWR_DEV_ON -> setTimSwt ->
Swt_BS_ON_MAIN_ON -> Swt_PYRO_PRE_MAIN_ON -> Swt_PYRO_PWR_ON ->
resTimSwt -> SWITCHDEV)
|~| (Com_PYRO_PWR_DEV_OFF -> setTimSwt ->
Swt_PYRO_PWR_MAIN_OFF -> resTimSwt -> SWITCHDEV)
|~| ... )
TIMCHK = elaTimSwt -> errorSwitchTimer -> TIMCHK
TCTIM = Tau_nextTC -> setTimTick -> elaTimTick -> TCTIM
```

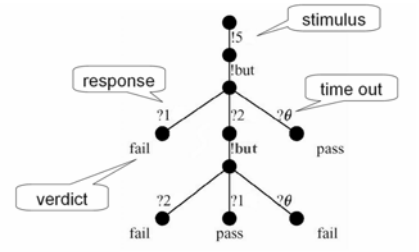
- Erzeugung von Tests aus der Zustandsübergangsrelation

Anderer Formalismus: Petrinetze



Testgenerierung

- Konformanzbegriff
  - Implementierung I ist konform zu einer Spezifikation S, wenn für alle Sequenzen σ von Ein/Ausgaben gilt:
    Beobachtungen (I nach σ) ⊆ Beobachtungen (S nach σ)
- Testfall:
  - deterministisches endliches Transitionssystem
  - Endzustände pass oder fail
  - von jedem Zustand eine Ausgabe- oder beliebige Eingabemöglichkeit



- Problem: Deadlock in der Implementierung
  - Modellierung durch Timeout-Signal

Generierungsalgorithmus

- Konstruiere den „Suspensionsautomaten"
  - ggf. implizite oder symbolische Repräsentation
  - ggf. „on-the-fly"-Verfahren
- Konstruiere die erzeugte Sprache durch rekursive Anwendung der Fälle
  - sende eine Ausgabe an das SUT
  - empfange und beurteile Eingabe vom SUT
    - unerwartet: fail
    - erwartet: inconc (pass oder rekursiver Aufruf)
    - don't care: pass
- Problem: wann aufhören?

Verschiedene Überdeckungsbegriffe (1)

Kontrollflussorientiert

- **Zustandsüberdeckung:** jeder Platz / Teilzustand wird von einem Testfall erfasst
- **Konfigurationsüberdeckung:** jeder erreichbare Globalzustand (Markierung) wird erfasst
- **Transitionsüberdeckung:** jeder Zustandsübergang wird erfasst
- **Pfadüberdeckung:** jeder Teilpfad einer gewissen Länge / bis zu einer gewissen Tiefe wird erfasst

Verschiedene Überdeckungsbegriffe (2)

Datenorientiert

- **Definitionsüberdeckung:** für jede definierte Variable wird die Verwendung von einem Testfall erfasst
- **Verwendungsüberdeckung:** für jede definierte Variable wird jede mögliche Verwendung von einem Testfall erfasst
- **Wertebereichsüberdeckung:** für jede definierte Variable werden alle Randwerte von einem Testfall erfasst

Generierung durch Modellprüfung

- Notiere das Modell in einer für den Modellprüfer geeigneten Form
- Notiere die gewünschte Überdeckung als Formel (z.B. Zustand xy ist *nicht* erreichbar)
- Der Gegenbeispielmechanismus des Modellprüfers liefert den Testfall

Für StateCharts, SMV und CTL einfach zu implementieren

Tools für modellbasierte Tests

Kriterien:

- unterstützte Modellierungssprachen, Effizienz
- Beeinflussbarkeit der Testfallerzeugung
- Ausführungsunterstützung
  - im Modell,
  - für einen virtuellen Prototyp, und
  - für ein reales SUT (**S**ystem **U**nder **T**est)

verfügbare Tools

- **Conformiq:** Generierung und Ausführung von Testfällen aus UML StateCharts, Übersetzung nach TTCN3
- **Agedis:** Murphi-Modelle mit Generierungsdirektiven (Projektionen)
- **TGV:** Lotos und SDL nach TTCN, Telekommunikation
- **Telelogic Tau:** generiert TTCN aus SDL
- **ASML:** Microsoft Abstract State Machine Language für .Net
- **RT-Tester:** RT-CSP und Timed Automata mit verteilter Ausführung, Zufallsüberdeckungen
- **UniTesK:** Modelle in J@va und C@++ (pre- und postconditions im Code: „grey box testing")

aktuelle Themen

Parallelität in Testfällen

- erwähnte Vorgehensweise kann ineffizient sein: exponentieller Faktor bei der Sequentialisierung eines parallelen Modells und Verteilung der erzeugten Sequenzen
- Wenn die Reaktionen des SUT parallel beobachtbar sind, kann die Parallelität im Modell erhalten bleiben
- Partialordnungsanalyse unter bestimmten Annahmen möglich

weitere Themen und Probleme

- Faktorisierung und Homomorphismen
- SUT Anbindung
- Testbeschreibungssprachen
- spezifikationsbasierte Testgenerierung
- Testorakel
- Realzeit und Lasttestsysteme
- Test und Fehlerinjektion

Vielen Dank für Ihre Aufmerksamkeit!