Testautomatisierung I

Prof. Dr. Jan Peleska
Universität Bremen – TZI
jp@tzi.de

2010-02-06

Revision: 1.14

Übersicht

- **Vorgehensmodelle:** Der Testprozess im Software-Engineering
- **Testarten** auf unterschiedlichen Systemebenen: Unittest – Integrationstest – Subsystemtest – Systemtest
- **Test-Techniken:** statische und dynamische Tests, Testabdeckung, Metriken
- **Spezifikationsbasiertes Testen** am Beispiel von Business Applications und Embedded Systems
- **Strukturelles Testen** gegen Anforderungs- und Architekturmodelle, sowie gegen Code-Kontrollstrukturen
- **Tools** und Auswahlkriterien

- **Design** for Testability

Vorgehensmodelle: Der Testprozess im Software-Engineering

Vorgehensmodelle (V-Modelle) ...

- ...regeln den Entwicklungsprozess durch Definition von **Aktivitäten** und **Produkten**, die in den einzelnen Entwicklungsphasen durchzuführen bzw. herzustellen sind,
- ...geben Hinweise über die mögliche Verteilung der Verantwortlichkeiten bei der Durchführung der Aktivitäten,
- ...sind **generische** Regelungen, die für das spezifische Entwicklungsvorhaben instantiiert werden **(Tayloring)**.

Vorgehensmodelle mit Softwarebezug – Beispiele

- **DIN ISO 9000 Teil 3** schreibt **Testen und Validierung** als festen Bestandteil des Qualitätssicherungssystems vor.
- **IEEE Std. 829-1998 – IEEE Standard for Software Test Documentation** ist ein Standard für Softwaretestdokumentation.
- **ISO/IEC 12119 – Information Technology – Software packages – Quality requirements and testing** ist ein sehr allgemein gehaltener Standard, der die allernotwendigsten Anforderungen an den Software-Testprozess beschreibt – kein Bezug zum Systemtesten

- **RTCA/DO-178B** (V-Modell für Softwareentwicklung in der zivilen Luftfahrt) unterscheidet zwischen den Prozessen **Verifikation** (aktive Durchführung von Verifikation, Validation und Test) und **Qualitätsmanagement** (Management aller Aktivitäten mit Qualitätsbezug).

- **V-Modell des Bundesinnenministeriums**[1] sieht Testen als Bestandteil der **Produktprüfung (Product Assessment)**, welches eine Aktivität der **Qualitätssicherung (Quality Assurance)** darstellt.

[1] siehe z.B. `http://www.informatik.uni-bremen.de/~uniform/gdpa/`

Methoden des Product Assessment nach V-Modell

- Statische Analyse,
- Test,
- Simulation,
- Korrektheitsbeweis,
- Symbolische Programm-(Spezifikations-)ausführung,
- Review,
- Inspektion.

Grundforderungen des V-Modells zum Testen – Testfall

- **Testfallbeschreibung:**
  - Was wird geprüft (*Test Objective*, Referenz zu Systemanforderungen, SW-Anforderungen, Entwurfsanforderungen, Benutzerhandbuch, Installationshandbuch ...)?
  - Was sind die Randbedingungen/Anfangsbedingungen für den Test (*Execution Condition*)?
  - Welche Eingabedaten sind für den Test erforderlich (*Inputs*) ?
  - Was sind die erwarteten Resultate (*Outputs*, *Expected Results*)?

Grundforderungen des V-Modells zum Testen – Testprozedur

- **Testprozedur:**
  - Instruktion ("Rezept") zur Ausführung eines Testfalls oder einer Kollektion mehrerer Testfälle
  - Instruktionen zur Erzeugung der konkreten Testdaten
  - Instruktionen zu Reihenfolge und Zeitpunkten (ggf. abhängig von den Ausgaben des Testlings), in denen die Testdaten auf die Eingabeschnittstellen des Testlings geschrieben werden
  - Instruktionen zur Auswertung der konkreten Ausgaben des Testlings
  - Instruktionen zum Vergleich der Ausgaben des Testlings gegen die erwarteten Resultate

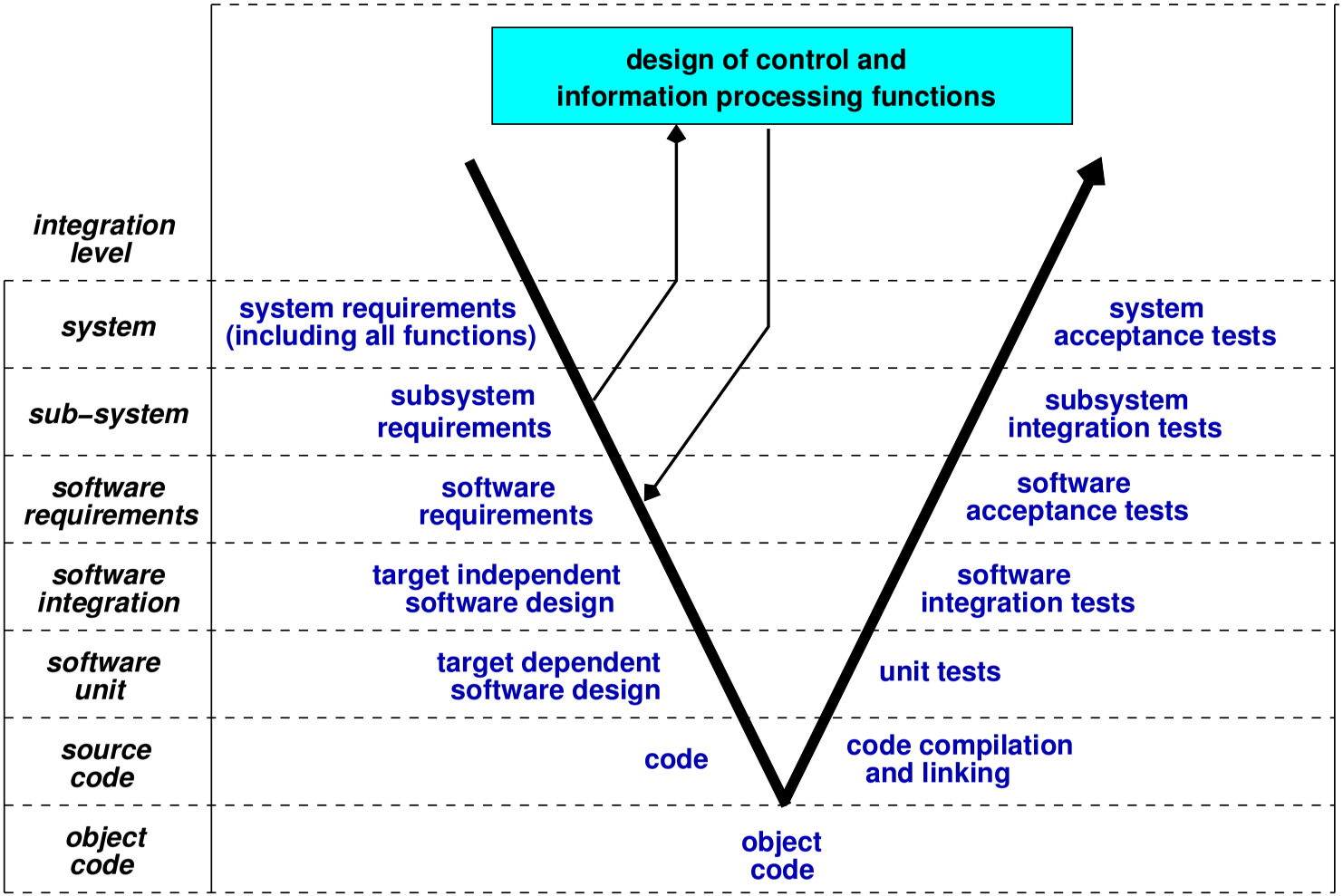
Grundforderungen des V-Modells zum Testen – Rückverfolgbarkeit (Traceability)

- **Überdeckungsmatrix:**
  - **Strukturüberdeckung:** Welche Architekturkomponenten und Codefragmente werden durch den Test berührt ?
  - **Anforderungsüberdeckung:** Welche Nutzer-/Sicherheits-/technischen Anforderungen, werden getestet ?

(Im Test zu prüfende) Qualitätseigenschaften – Gliederung

- **Wirksamkeit:** Eignung des Systems, seine intendierte Aufgabe zu erfüllen. Die Prüfung der Wirksamkeit durch Tests, Analysen, Inspektionen etc. heisst **Validierung**.
- **Korrektheit:** Übereinstimmung des Systems mit spezifizierten Eigenschaften. Die Prüfung der Korrektheit durch Tests, Analysen, Inspektionen, Formale Verifikation etc. heisst **Verifikation**.

Zuordnung: Testaktivitäten im Entwicklungsprozess



(Im Test zu prüfende) Systemeigenschaften – Gliederung

- **Funktionale Eigenschaften:** Charakteristika der
  - Daten
  - Datentransformationen
  - Kausaleigenschaften (auch "Kontrolleigenschaften", d. h. Reihenfolge bestimmmter Ereignisse, Synchronisation)
  - Zeitverhalten:
    - Diskrete Zeitpunkte für die Änderung von Daten (*time-discrete behaviour*)
    - Kontinuierliche Änderung von Daten (Stellgrößen) über der Zeit (*time-continuous behaviour*)

(Im Test zu prüfende) Systemeigenschaften – Gliederung

- **Struktureigenschaften:**
  - Eigenschaften der **System- und Softwarearchitektur:** Aufrufhierarchie zwischen Prozeduren oder Funktionen – Nachrichtenaustausch zwischen Objekten – Datenfluss zwischen Threads, Prozessen, Rechnern – Zugriff auf globale Daten – Kapselung von Methoden in Klassen – Kapselung von Prozeduren und Funktionen in Threads oder Prozessen – Hardwarearchitektur
  - Eigenschaften der **Softwarekontrollstrukturen** innerhalb eines Moduls: `;` `-` `if-then-else` `-` `while`

(Im Test zu prüfende) Systemeigenschaften – Gliederung

- **nicht-funktionale Eigenschaften:**
  - Dependability: Reliability – Availability – Safety – Security
  - Wartbarkeit
  - Betriebskosten
  - Ergonomische Eigenschaften – Benutzerfreundlichkeit
  - Lastverhalten – Performanz – vom (Daten-)Volumen abhängiges Verhalten – Stressverhalten
  - Robustheit
  - Kompatibilität
  - Konfigurationseigenschaften (zulässige Betriebssystemversionen, Sprachauswahl,...)
  - Dokumentation

Testarten – Gliederung

Die Testarten lassen sich nach folgenden Kriterien klassifizieren:

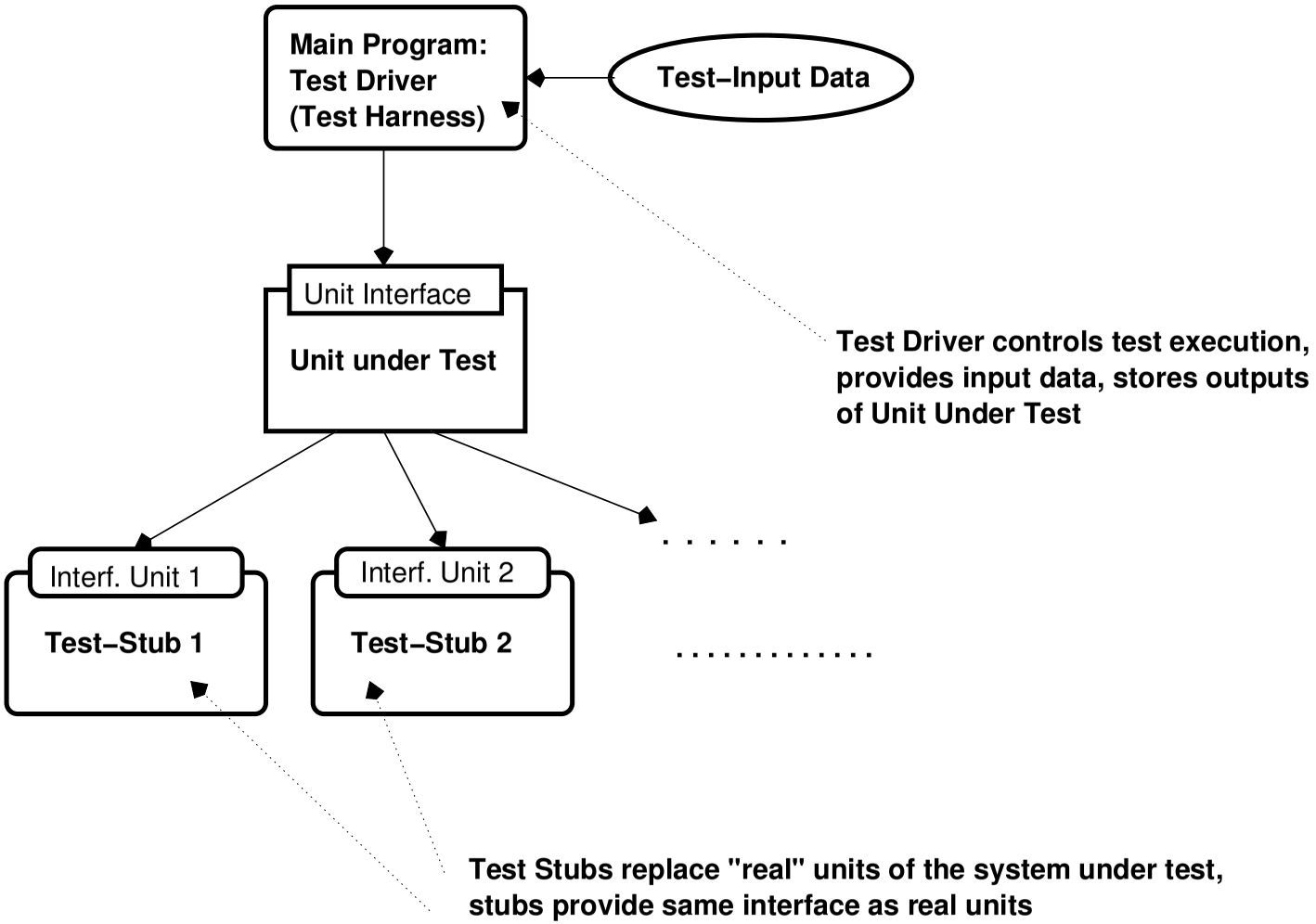
- **Systemeigenschaften:** Funktionale Tests – Strukturtests – nicht funktionale Tests (siehe oben)
- **Integrationslevel:** Unittests – Integrationstests – Subsystemtests – Systemtests
- **Umgebungsbedingungen:** Test des Normalverhaltens – Test des Ausnahmeverhaltens
- **Beobachtungstiefe:** White-Box Tests – Black-Box Tests

- **Intrusive/nicht intrusive Tests:** Intrusive Tests verändern den Testling vor oder während der Testausführung (z. B. Hardwaremodifikationen, Crash Tests, instrumentierter Softwarecode). Nicht intrusive Tests verändern den Testling nicht.

Testarten – Unittests (= Modultests)

- **Testscope:** Unittests untersuchen die kleinsten "Bausteine" des Softwaresystems: Functions, Threads, Objects, Data Container ...
- **Testziele:** Korrekte Arbeitsweise der Unit in Abwesenheit anderer zum System gehörigen Softwarekomponenten $\Longrightarrow$ die Unit wird isoliert geprüft.
- **Referenz:** Modulspezifikation, Feinentwurf

Testumgebung für Unittests



Testarten – Software-Integrationstest

- **Testscope:** Teilsysteme kooperierender Komponenten (Functions, Threads, Processes)
- **Testziele:** Korrekte Kooperation der Komponenten prüfen:
  - Datenfluss, Schnittstellen
  - Synchronisation beim Zugriff auf gemeinsame Ressourcen
- **Referenz:** Software-Subsystemanforderungen, Modulspezifikationen, SW-Architekturdesign (Grobentwurf)

Testarten – Subsystem-Integrationstest

- **Testscope:** HW/SW-Subsysteme (einzeln und kooperierend) – endet mit dem Test des Gesamtsystems
- **Testziele:** Korrekte Arbeitsweise der Subsysteme prüfen:
  - Zusammenspiel zwischen Hardware und Software im einzelnen Subsystem
  - Kooperation mehrerer Subsysteme
  - Funktionsweise der Kommunikationsmedien zwischen HW/SW-Subsystemen
- **Referenz:** Systemanforderungen, Subsystemanforderungen, Systemdesign

Testarten – Systemannahmetest

- **Testscope:** Gesamtsystem
- **Testziele:** Nachweis der spezifizierten Benutzeranforderungen $\Longrightarrow$ Systemannahmetest ist häufig eine Untermenge der auf Gesamtsystem-Level durchgeführten Integrationstests.

  Früher wurden Systemannahmetests auch mit dem Ziel der Validierung (d.h. Prüfung des Systems hinsichtlich seiner Wirksamkeit) durchgeführt. Eine so späte Validierung wird heute als viel zu riskant für den Projekterfolg angesehen.
- **Referenz:** Benutzeranforderungen

Testarten – Normalverhalten versus Ausnahmeverhalten

- **Normal Behaviour Tests** prüfen die Korrektheit des Testlings unter normalen Umgebungsbedingungen.
- **Exceptional Behaviour Tests (Robustness Tests)** prüfen die Korrektheit des Testlings in spezifizierten (d. h. vorhersehbaren) Ausnahmesituationen.
- Normal und Exceptional Behaviour Tests können auf jeder Integrationsstufe (Unit-Level bis System-Level) durchgeführt werden.

Testziele der Robustheitstests

Test-Level	Testziele – Robust gegen ...
Unit	falsche Eingabeparameter – falsche Datenobjekte
Integration	falsche Daten an Prozess/Thread-Interfaces – falsche I/O-Folgen in der Kommunikation zwischen Komponenten – fehlende Ressourcen – Timeouts – Ausfall einzelner SW-Komponenten
Subsystem	fehlerhafte Kommunikationsprotokolle – lokale Hardwarefehler – lokale Überlast – Ausfall einzelner SW-Komponenten
System	Subsystemausfälle – Überlastsituationen im Gesamtsystem

Testarten – Black-Box versus White-Box

- **Black-Box Tests** untersuchen funktionale Eigenschaften der Komponente durch Analyse ihres **an der Schnittstelle sichbaren** Verhaltens.

  Black-Box Tests können i.a. nicht alle relevanten funktionalen Eigenschaften des Testlings prüfen ($\Longleftarrow$ nicht sichtbare interne Entscheidungen führen zu nicht-deterministischem Verhalten an der Schnittstelle)!

  - $\Longrightarrow$ erzielte Strukturüberdeckung beim Black-Box Test messen !
  - $\Longrightarrow$ ggf. zusätzliche White-Box Tests zum Erreichen der Strukturüberdeckung durchführen !

Testarten – Black-Box versus White-Box

- **White-Box Tests** untersuchen strukturelle Eigenschaften der Komponente.

  Für sinnvolle White-Box Tests müssen die Ergebnisse auch bzgl. ihrer funktionalen Korrektheit geprüft werden!

- Black-Box und White-Box Tests können auf jeder Integrationsstufen (Unit-Level bis System-Level) durchgeführt werden.

Test-Techniken

- Statische Tests – formale Verifikation
- Dynamische Tests
- Testabdeckung
- Metriken

Spezifikationsbasiertes Testen

- **Spezifikationsbasiertes Testen auf Grundlage formaler – maschinell interpretierbarer – Spezifikationen ist die Voraussetzung für Automatisierung!**

Spezifikationsbasiertes Testen

Einige Fakten aus der Theorie:

- Anforderungen, die sich mit Hilfe paralleler, kooperierender Zustandsmaschinen formulieren lassen, erlauben automatische Testerzeugung, Durchführung und Auswertung in Echtzeit.
- Auch für nicht terminierende Systeme (z.B. Controller) lässt sich die Vollständigkeit der erzielten Überdeckung prüfen, wenn eine obere Schranke für die Anzahl der möglichen Systemzustände bekannt ist.
- Es gibt Anforderungen, die untestbar sind.
- Es gibt Anforderungen, die nicht automatisiert testbar sind.

Spezifikationsbasiertes Testen – Klassifikation

Die Zielsetzungen spezifikationsbasierter Tests lassen sich folgendermaßen klassifizieren:

1. Nachweisziel **Modelläquivalenz**: Tests sollen nachweisen, dass das SUT ein zum Spezifikationsmodell **äquivalentes** Verhalten zeigt. Die gebräuchlichste Äquivalenzrelation ist **Bisimilarität**.

Spezifikationsbasiertes Testen – Klassifikation

2. Nachweisziel **Modellverfeinerung**: Tests sollen nachweisen, dass das SUT-Verhalten das Modellverhalten **verfeinert**. Es wir damit keine strikte Äquivalenz mehr gefordert sondern "Gleichwertigkeit oder Verbesserung des SUT Verhaltens gegenüber der Spezifikation". Die gebräuchlichsten Verfeinerungsrelationen sind **Failures-** oder **Failures-Divergence**-refinement.

   Wenn Divergenzfreiheit bereits gesichert ist und asynchrone Kommunikation verwendet wird, stimmen die Verfeinerungsbegriffe mit Bisimilarität überein.

Spezifikationsbasiertes Testen – Klassifikation

3. Nachweisziel **Abwesenheit bestimmter Fehlertypen**: Es werden Tests entwickelt, welche zwar nicht Modelläquivalenz oder Modellverfeinerung, aber wenigstens die Abwesenheit bestimmter Fehlertypen nachweisen.

Spezifikationsbasiertes Testen – Nachweis der Modelläquivalenz

Die klassische W-Methode nach Tsun S. Chow

Literaturangabe: Tsun S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering* SW-4, No. 3, pp. 178-187(1978).

Zielsetzung: Weise Äquivalenz von Spezifikation A und Implementierung B durch eine endliche Anzahl von Testfällen nach, die "mechanisch" aus A konstruiert werden können.

Anwendungsgebiet – Voraussetzung: Spezifikation A und Implementierung B können als (deterministische!) Mealy-Automaten implementiert werden.

Spezifikationsbasiertes Testen – W-Methode

Voraussetzungen, Bezeichnungen:

- A und B sind über demselben Alphabet $\Sigma = I \cup O$ definiert; I enthält die Eingabesymbole, O die Ausgabesymbole.
- Die Transitionsfunktionen $\delta_A : Q(A) \times I \to Q(B) \times O$ und $\delta_B : Q(B) \times I \to Q(B) \times O$ sind totale Funktionen.
- Für $\delta(q_1, x) = (q_2, y)$ schreiben wir $q_1 \xrightarrow{x/y} q_2$.
- Führt die Eingabefolge $p = \langle x_1, \ldots, x_k \rangle$ von Zustand q_1 in Endzustand q_2, schreiben wir $q_1 \stackrel{p}{\Longrightarrow} q_2$.
- A und B sind minimal (diese Voraussetzung kann tatsächlich abgeschwächt werden).

Spezifikationsbasiertes Testen – W-Methode

- Die Zustandsmenge $Q(A)$ hat n Zustände, $Q(B)$ hat m Zustände, $m \geq n$; Anfangszustände sind q_A, q_B.
- **Testfälle** sind Input-Traces $p \in I^*$.
- Als **Testorakel** dient die Spezifikation A: Eine beobachtete I/O-Trace $u \in \Sigma^*$ kann automatisiert überprüft werden, ob sie ein gültiges Wort der durch A geprüften Sprache darstellt.
- Eine Menge $P \subseteq I^*$ heisst **Transitionsüberdeckung (Transition Cover)** zu A, wenn gilt:

$$\forall q_1 \xrightarrow{x/y} q_2 \in \delta_A : \exists p \in P : q_A \stackrel{p}{\Longrightarrow} q_1 \wedge p \frown \langle x \rangle \in P$$

Spezifikationsbasiertes Testen – W-Methode

- Eine Menge $W \in I^*$ heisst **Charakterisierungsmenge (Characterisation Set)** von A wenn für alle $q_1, q_2 \in Q(A)$ ein $w \in W$ existiert, welches die beiden Zustände **unterscheidet**, d. h., die Inputs aus w erzeugen vom Zustand q_1 ausgehend eine andere Ausgabefolge als vom Zustand q_2 ausgehend.
- Definiere $X^n = \{p \in I^* \mid \#p = n\}$ für $n \geq 0$.
- Definiere $U_1 \cdot U_2 = \{u_1 \frown u_2 \mid u_i \in U_i, i = 1, 2\}$ für $U_1, U_2 \subseteq I^*$.
- Definiere die Menge $\mathcal{W}(A)$ der **W-Test Cases** von A durch

$$\mathcal{W}(A) = P \cdot (\bigcup_{i=0}^{m-n} (X^i \cdot W))$$

Spezifikationsbasiertes Testen – W-Methode

Chows Theorem: Wenn B alle W-Test Cases $\mathcal{W}(A)$ besteht – d. h., B liefert für alle $w \in \mathcal{W}(A)$ dieselben Ausgabefolgen wie A – sind A und B äquivalent.

Hinweise:

- Äquivalenz bedeutet **Sprachäquivalenz** im Sinne der Automatentheorie
- Da A und B deterministisch sind, ist Äquivalenz gleichwertig zu **Bisimilarität**, d. h., es existiert eine bijektive Abbildung $f : Q(A) \longrightarrow Q(B)$ so dass

$$\forall q_1, q_2 \in Q(A) : q_1 \xrightarrow{x/y} q_2 \implies f(q_1) \xrightarrow{x/y} f(q_2)$$

  f heisst **Isomorphismus** zwischen A und B.

Spezifikationsbasiertes Testen – Definitionen zur W-Methode

Definition 1: Sei $V \subseteq I^*$ eine Menge von Eingabefolgen.

1. Zwei Zustände $q_i \in Q(A), q_j \in Q(B)$ heissen **V-äquivalent** ($q_i \sim_V q_j$), wenn jede Eingabefolge $p \in V$ von q_i in A ausgehend dieselben Ausgaben erzeugt, wie von q_j in B ausgehend.

2. Die Automaten A und B heissen **V-äquivalent** ($A \sim_V B$), wenn $q_A \sim_V q_B$, d. h., die Anfangszustände sind V-äquivalent.

Offenbar ist $\sim_V$ eine Äquivalenzrelation auf den Zuständen von A bzw. B.

Spezifikationsbasiertes Testen – Beweis der W-Methode

Offenbar gilt für bisimilare Automaten (geschrieben $A \approx B$):

$$A \approx B \implies (\forall V \subseteq I^* : A \sim_V B)$$

Chows Theorem kann mit diesen Definitionen folgendermaßen umformuliert werden:

Chows Theorem – Variante 2: $A \sim_{\mathcal{W}(A)} B \implies A \approx B$

Der Beweis ergibt sich aus den folgenden Lemmata. Dabei nehmen wir an, dass A n Zustände und B $m \geq n$ Zustände haben und beide minimal sind. W bezeichnet die Charaktersierungsmenge von A.

Spezifikationsbasiertes Testen – Beweis der W-Methode

Lemma 1: Angenommen, die Menge W (Charaktersierungsmenge von A), partitioniert $Q(B)$ in mindestens n Äquivalenzklassen. Dann partitioniert $Z = \bigcup_{i=0}^{m-n}(X^i \cdot W)$ die Zustandsmenge $Q(B)$ in m Klassen, das heisst, je zwei Zustände aus $Q(B)$ sind durch $\mathcal{W}(A)$ unterscheidbar.

Beweis: Definiere $Z(\ell) = \bigcup_{i=0}^{\ell}(X^i \cdot W)$. Offenbar ist $Z(m-n) = Z$. Führe folgenden Induktionsbeweis für $\ell = 0, 1, \ldots m-n$ durch:

$Z(\ell)$ partitioniert $Q(B)$ in $\ell + n$ Klassen $\qquad (*)$

Mit $\ell = m - n$ folgt hieraus die Aussage des Lemmas.

Spezifikationsbasiertes Testen – Beweis der W-Methode

Beweis von (∗) – Induktionsverankerung: Für $\ell = 0$ stimmt (∗) mit der Voraussetzung des Lemmas überein.

Induktionsvoraussetzung: Für gegebenes $\ell \in \{0, 1, \ldots m - n - 1\}$ teilt $Z(\ell)$ die Zustandsmenge $Q(B)$ in mindestens $\ell + n$ Klassen.

Induktionsschritt: Wir zeigen, dass $Z(\ell + 1)$ die Zustandsmenge $Q(B)$ in mindestens $\ell + n + 1$ Klassen teilt.

Wenn $Z(\ell)$ bereits die Zustandsmenge $Q(B)$ in mindestens $\ell + n + 1$ Klassen teilt, ist nichts zu zeigen. Andernfalls muss ein $k > \ell$ existieren, so dass gilt (beachte $Z(k) = Z(k-1) \cup X^k \cdot W$)

$$\exists r_1, r_2 \in Q(B) : r_1 \sim_{Z(k-1)} r_2 \wedge r_1 \not\sim_{(X^k \cdot W)} r_2$$

Spezifikationsbasiertes Testen – Beweis der W-Methode

Wenn $k = \ell + 1$ ist nichts weiter zu zeigen, denn $(*)$ gilt für $Z(k) = Z(\ell + 1)$.

Andernfalls $(k \geq \ell + 2)$ sei $p = \langle x_1, \ldots, x_k \rangle \frown w, w \in W$ die r_1 und r_2 unterscheidende Eingabesequenz.

Seien r_1', r_2' so gewählt, dass $r_1 \overset{\langle x_1, \ldots x_{k-\ell-1} \rangle}{\Longrightarrow} r_1'$, $r_2 \overset{\langle x_1, \ldots x_{k-\ell-1} \rangle}{\Longrightarrow} r_2'$.
Dann sind r_1', r_2' gerade durch $Z(\ell + 1)$ unterscheidbar. $\square$

Spezifikationsbasiertes Testen – Beweis der W-Methode

Lemma 2: Gegeben $Z = \bigcup_{i=0}^{m-n}(X^i \cdot W)$ wie in Lemma 1 eingeführt. Es gilt $A \approx B$ genau dann, wenn folgende Bedingungen erfüllt sind:

1. Für alle $a \in Q(A)$ existiert $b \in Q(B)$ mit $a \sim_Z b$.
2. Die Anfangszustände sind äquivalent: $q_A \sim_Z q_B$.
3. Für alle $a_i \xrightarrow{x/y} a_j$ in A existiert $b_i, b_j \in Q(B)$, so dass $a_i \sim_Z b_i$, $a_j \sim_Z b_j$ und $b_i \xrightarrow{x/y} b_j$.

Spezifikationsbasiertes Testen – Beweis der W-Methode

Beweis (a): Wenn $A \approx B$, folgt $(1 — 3)$ aus der Definition der Äquivalenz: Die Eigenschaften werden direkt durch die Existenz der Isomorphie $f : Q(A) \longrightarrow Q(B)$ impliziert.

Beweis (b): Angenommen, es gelten die Eigenschaften $(1 — 3)$. Wir müssen zeigen, dass hieraus die Existenz eines Isomorphismus $f : Q(A) \longrightarrow Q(B)$ folgt. Dazu wollen wir nachzuweisen, dass die Funktion

$$f(q_A) = q_B$$
$$(q_A \overset{\langle x_1,\ldots,x_\ell \rangle}{\Longrightarrow} a \wedge q_B \overset{\langle x_1,\ldots,x_\ell \rangle}{\Longrightarrow} b) \Longrightarrow f(a) = b$$

wohldefiniert, injektiv und surjektiv ist. Aus (3) folgt dann ausserdem, dass auch $\forall a \in Q(A) : a \sim_Z f(a)$ gilt.

Spezifikationsbasiertes Testen – Beweis der W-Methode

Wohldefiniertheit: Wir müssen zeigen, dass für *verschiedene* Eingabefolgen $q_A \overset{\langle x_1,\ldots,x_\ell\rangle}{\Longrightarrow} a$, $q_A \overset{\langle x'_1,\ldots,x'_k\rangle}{\Longrightarrow} a$, die zum selben Nachzustand a in A führen, auch immer der selbe Nachzustand in B erreicht wird. Gelte also $q_B \overset{\langle x_1,\ldots,x_\ell\rangle}{\Longrightarrow} b$ und $q_B \overset{\langle x'_1,\ldots,x'_k\rangle}{\Longrightarrow} b'$. Wir müssen $b = b'$ zeigen.

Wegen (3) gilt $a \sim_Z b \sim_Z b'$.

Wir zeigen jetzt, dass Z je 2 Zustände des Automaten B unterscheidet, so dass aus $b \sim_Z b'$ auch $b = b'$ gefolgert werden kann. Damit ist dann die Wohldefiniertheit von f nachgewiesen.

Spezifikationsbasiertes Testen – Beweis der W-Methode

Z unterscheidet je 2 Zustände von B: Die Charakterisierungsmenge W von A teilt $Q(A)$ in $n = card(Q(A))$ Klassen. Aus (1) und (3) folgt, dass W auch $Q(B)$ in mindestens n Klassen partitioniert:

Angenommen a_1 und a_2 werden von $w \in W$ unterschieden. Gelte $q_A \overset{\langle x_1,\ldots,x_\ell \rangle}{\Longrightarrow} a_1$ und $q_A \overset{\langle x'_1,\ldots,x'_k \rangle}{\Longrightarrow} a_2$. Diese beiden Eingabefolgen führen uns nach (3) auf Zustände $b_1, b_2 \in Q(B)$, so dass $a_i \sim_Z b_i, i = 1, 2$ gilt.

Spezifikationsbasiertes Testen – Beweis der W-Methode

Weil $W \subseteq Z$ und wieder wegen (3) muss $b_1 \stackrel{w}{\Longrightarrow}$ dieselben Ausgaben erzeugen wie $a_1 \stackrel{w}{\Longrightarrow}$ und $b_2 \stackrel{w}{\Longrightarrow}$ dieselben Ausgaben wie $a_2 \stackrel{w}{\Longrightarrow}$. Da w von a_1 bzw. a_2 ausgehend unterschiedliche Ausgaben erzeugt, muss dies also auch für $b_1 \stackrel{w}{\Longrightarrow}$ und $b_2 \stackrel{w}{\Longrightarrow}$ gelten. Eingabefolge w unterscheidet also auch b_1 und b_2, d.h. $b_1 \neq b_2$.

Da $W \subseteq Z$ und W die Zustände von B in mindestens n Klassen teilt, können wir Lemma 1 anwenden: Z unterscheidet *alle* Zustände von B. Gegeben sei $b \in Q(B)$. Aus $b \sim_Z b'$ folgt also $b = b'$, was für die Wohldefiniertheit von f zu zeigen war.

Spezifikationsbasiertes Testen – Beweis der W-Methode

Injektivität: Sei $a_i \in Q(A), i = 1, 2, a_1 \neq a_2$ und $b_i = f(a_i) \in Q(B)$. Wir müssen zeigen, dass $b_1 \neq b_2$ gilt.

Da $a_1 \not\sim_W a_2$ und $W \subseteq Z$, folgt $a_1 \not\sim_Z a_2$. Aus (3) folgt $a_i \sim_Z f(a_i) = b_i, i = 1, 2$ und folglich $b_1 \not\sim_Z b_2$, also auch $b_1 \neq b_2$.

Spezifikationsbasiertes Testen – Beweis der W-Methode

Surjektivität: Gegeben $b \in Q(B)$ und Eingabefolge, so dass $q_B \overset{\langle x_1,\ldots,x_\ell\rangle}{\Longrightarrow} b$. Da A und B deterministisch sind, sind Nachzustände $b \in Q(B), a \in Q(A)$ durch $q_B \overset{\langle x_1,\ldots,x_\ell\rangle}{\Longrightarrow} b$ und $q_A \overset{\langle x_1,\ldots,x_\ell\rangle}{\Longrightarrow} a$ eindeutig bestimmt. Da wir bereits wissen, dass f wohldefiniert ist, folgt $f(a) = b$ aus der Definition von f. $\square$

Spezifikationsbasiertes Testen – Beweis der W-Methode

Lemma 3: $A \sim_{P \cdot Z} B$ genau dann, wenn

1. Für alle $a \in Q(A)$ existiert $b \in Q(B)$ mit $a \sim_Z b$.
2. Die Anfangszustände sind äquivalent: $q_A \sim_Z q_B$.
3. Für alle $a_i \xrightarrow{x/y} a_j$ in A existiert $b_i, b_j \in Q(B)$, so dass $a_i \sim_Z b_i$, $a_j \sim_Z b_j$ und $b_i \xrightarrow{x/y} b_j$.

Hinweis: Da (1 — 3) genau die Voraussetzungen von Lemma 2 sind und gemäß Lemma 2 die Äquivalenz von A und B implizieren, ist mit Lemma 3 folglich Chow's Theorem bewiesen, denn es gilt dann

$$A \sim_{P \cdot Z} B \Leftrightarrow A \approx B$$

Spezifikationsbasiertes Testen – Beweis der W-Methode

Beweis von Lemma 3 – (a): Angenommen es gelten (1—3). Dann folgt aus Lemm 2 $A \approx B$, und dies impliziert trivialerweise $A \sim_{P \cdot Z} B$.

Beweis von Lemma 3 – (b): Angenommen, $A \sim_{P \cdot Z} B$. Gegeben $a \in Q(A)$ und Eingabefolge $p \in P$ mit $q_A \stackrel{p}{\Longrightarrow} a$. Diese Folge p existiert wegen der Transitionsüberdeckungseigenschaft von P.

Da A und B deterministisch sind, ist b durch $q_B \stackrel{\langle x_1,\ldots,x_\ell \rangle}{\Longrightarrow} b$ eindeutig bestimmt. Da $q_A \sim_{P \cdot Z} q_B$ und $p \in P$ folgt $a \sim_Z b$. Damit haben wir (1) und (2) (wegen $\langle\ \rangle \in P$) gezeigt.

Spezifikationsbasiertes Testen – Beweis der W-Methode

Sei $a_1 \xrightarrow{x/y} a_2$ eine Transition in A. Sei $p \in P$ mit $q_A \stackrel{p}{\Longrightarrow} a_1$. Nach Eigenschaft der Transitionsüberdeckung gilt folglich auch $p \frown \langle x \rangle \in P$. Seien $b_1, b_2 \in Q(B)$ durch $q_B \stackrel{p}{\Longrightarrow} b_1$ und $q_B \stackrel{p \frown \langle x \rangle}{\Longrightarrow} b_2$ eindeutig bestimmt (Determinismus).

Wegen $A \sim_{P.Z} B$ folgt $a_i \sim_Z b_i, i = 1, 2$. Weiterhin muss bei der Transition $b_1 \xrightarrow{x/y'} b_2$ $y' = y$ gelten, weil sonst a_1 und b_1 durch die Eingabe x unterschieden werden könnten – dies ergäbe einen Widerspruch zu $a_1 \sim_Z b_1$. □

Spezifikationsbasiertes Testen – BFS-Algorithmus zur Konstruktion der Transitionsüberdeckung

- Breitensuche über den deterministischen endlichen Automaten A (*Deterministic Finite Automaton DFA*) wie oben definiert.
- tc: Bei Rückkehr der Funktion tc enthält der (wie bei Pascal gleich benannte) Rückgabewert die Menge aller Eingabefolgen, welche die Transitionsüberdeckung ausmachen.
- α ist die übliche Queue, die bei BFS-Algorithmen verwendet wird.
- N ist eine Hilfsmenge von Knoten des Automaten A, die nicht mehr in die Queue α eingetragen werden sollen

- τ bildet Knoten q, die von denen aus der Graph von A noch weiter exploriert wird, auf die bisher konstruierte Eingabefolge ab, die in A vom Initialzustand q_A zu q führt.

function tc(**in** $A : DFA$) : $\mathbb{P}(I^*)$
begin
  $tc := \{\langle\ \rangle\};\ \alpha := \langle q_A \rangle;\ N := \{q_A\};\ \tau := \{q_A \mapsto \langle\ \rangle\};$
  while $0 < \#\alpha$ **do**
    $u = head(\alpha);$
    foreach $x \in I$ **do**
      $q := \delta_A(u, x);$
      $tc := tc \cup \{\tau(u) \frown \langle x \rangle\};$
      if $q \notin N$ **then**
        $N := N \cup \{q\};$
        $\tau := \tau \oplus \{q \mapsto \tau(u) \frown \langle x \rangle\};$
        $\alpha := \alpha \frown \langle q \rangle;$
      endif
    enddo
    $\alpha := tail(\alpha);$
  enddo
end

Spezifikationsbasiertes Testen – Algorithmus zur Konstruktion der Charakterisierungsmenge W

- Die Charakterisierungsmenge W ensteht automatisch bei einem Standardverfahren, welches zu gegebenem DFA A' den minimalen A konstruiert.
- Die Verwendung eines *minimalen* DFA als Referenzspezifikation ist vorteilhaft für die W-Methode, weil damit auch die Größe der Transitionsüberdeckung so klein wie möglich gehalten wird.

Spezifikationsbasiertes Testen – Algorithmus zur Konstruktion der Charakterisierungsmenge W

- Es sollte daher
  - zuerst die Charakterisierungsmenge W zu gegebener DFA-Spezifikation A' konstruiert und dabei auch gleich A' zu A reduziert werden (wenn A' bereits minimal ist, wird trotzdem W erzeugt),
  - danach die Transitionsüberdeckung konstruiert werden.
- Beim nachfolgenden Algorithmus w wird sinnvollerweise angenommen, dass $Q(A)$ keine unerreichbaren Zustände enthält.

Spezifikationsbasiertes Testen – Algorithmus zur Konstruktion der Charakterisierungsmenge W

- Wir nehmen an, dass der DFA *keinen* akzeptierenden Zustand hat; dies ist sinnvoll für Steuerungsanwendungen, bei denen Terminierung der Eingaben bzw. von A selbst nicht vorgesehen ist, und wo es aus Robustheitsgründen auch keine "unerlaubten" Eingabefolgen gibt, die zu einem fehlerhaften Abbrechen der Verarbeitung von A führen.
- Die Reduktion des DFA wir im letzten Schritt des Algorithmus vorgenommen, den man also auch ggf. weglassen kann, da zu diesem Zeipunkt die Charakterisierungsmenge bereits konstruiert ist.

Spezifikationsbasiertes Testen – Algorithmus zur Konstruktion der Charakterisierungsmenge W

Notation:

- $\omega_A : Q(A) \times I \longrightarrow O; \omega_A(q, x) = y \Leftrightarrow (\exists q' \in Q(A) : \delta_A(q, x) = (q', y))$ bildet *(Ausgangszustand,Eingabe)* auf die zugehörige Ausgabe y ab.
- $\lambda_A : Q(A) \times I \longrightarrow Q(A); \lambda_A(q, x) = q' \Leftrightarrow (\exists y \in O : \delta_A(q, x) = (q', y))$ bildet *(Ausgangszustand,Eingabe)* auf den zugehörigen Nachzustand q' ab.
- Wir nehmen an, dass alle Zustände $q, q' \in Q(A)$ eindeutig nummeriert sind, so dass Relation $<$ wohldefiniert ist und für $q \neq q'$ entweder $q < q'$ oder $q' < q$ gilt.

Spezifikationsbasiertes Testen – Algorithmus zur Konstruktion der Charakterisierungsmenge W

Notation:

- Auf Paaren $(q, q') \in Q(A) \times Q(A)$ definiert

$$od : Q(A) \times Q(A) \not\longrightarrow Q(A) \times Q(A)$$

$$od(q, q') = \begin{cases} (q, q') & \text{falls } q < q' \\ (q', q) & \text{falls } q' < q \end{cases}$$

eine Abbildung, welche paarweise verschiedene Zustände gemäß ihrer Ordnung sortiert.

Spezifikationsbasiertes Testen – Algorithmus zur Konstruktion der Charakterisierungsmenge W

Notation:

- Auf Eingabetraces $w, w' \in I^*$ schreiben wir $w < w'$, wenn w ein echtes Präfix von w' ist.
- Auf Paaren $(q, q') \in Q(A) \times Q(A)$ definiert $\beta : Q(A) \times Q(A) \not\longrightarrow I^*$ eine Abbildung, welche unterscheidbaren Zuständen eine entsprechende nicht leere Eingabefolge zuordnet, welche von q bzw. q' ausgehend mindestens eine unterschiedliche Ausgabe produziert.

Spezifikationsbasiertes Testen – Algorithmus zur Konstruktion der Charakterisierungsmenge W

procedure W(**inout** A : DFA, **inout** W : $\mathbb{P}(I^*)$)
begin
$D : \mathbb{P}(Q(A) \times Q(A))$; // Ordered distinguishable state pairs
$\beta : Q(A) \times Q(A) \not\longrightarrow I^*$; // Map elements from D to input trace
$D := \{\}$; $\beta := \{\}$;
// Initialisation: Insert all ordered pairs of states into D
// which can be distinguished by a single input
$distinguishedByOne(A, D, \beta)$;
// Identify all distinguishable state pairs, while constructing W
$generateW(A, D, \beta, W)$;
// Optionally, reduce the DFA
$reduceA(A, D, \beta)$;
end

Spezifikationsbasiertes Testen – Algorithmus zur Konstruktion der Charakterisierungsmenge W

procedure *distinguishedByOne*(**in** $A : \mathrm{DFA}$,
inout $D : \mathbb{P}(Q(A) \times Q(A))$,
inout $\beta : Q(A) \times Q(A) \not\longrightarrow I^*$)
begin
  foreach $p < q \in Q(A) \times Q(A)$ **do**
    foreach $x \in I$ **do**
      if $\omega_A(p, x) \neq \omega_A(q, x)$ **then**
        $D := D \cup \{(p, q)\}$;
        $\beta := \beta \oplus \{(p, q) \mapsto \langle x \rangle\}$;
      endif
    enddo
  enddo
end

Spezifikationsbasiertes Testen – Algorithmus zur Konstruktion der Charakterisierungsmenge W

```
procedure generateW(in A : DFA,
                    inout D : ℙ(Q(A) × Q(A)),
                    inout β : Q(A) × Q(A) ⇸ I*,
                    out W : ℙ(I*))
begin
   b : bool; b := false;
   do
      foreach p < q ∈ (Q(A) × Q(A)) − D do
         foreach x ∈ I do
            v := λ_A(p, x); z := λ_A(q, x);
            if od(v, z) ∈ D then
               b := true;
               w := ⟨x⟩ ⌢ β(od(v, z));
               //Remove traces which are prefixes of the new (longer) one
               foreach (p', q') ∈ D do
                  if β(p', q') < w then
                     β := β ⊕ {(p', q') ↦ w};
                  endif
               enddo
               β := β ⊕ {(p, q) ↦ w};
               D := D ∪ {(p, q)};
            endif
   while b;
   W := ran(β);
end
```

Spezifikationsbasiertes Testen – Algorithmus zur Konstruktion der Charakterisierungsmenge W

procedure *reduceA*(**inout** $A : \mathrm{DFA}$,
inout $D : \mathbb{P}(Q(A) \times Q(A))$)
begin
$A_r : DFA$;
// Definition of equivalence classes:
// $[p] = \{q \in Q(A) \mid od(p,q) \notin D\}$
// States of the minimised DFA are equivalence classes,
// each class represented by a state p of A which is
// member of a distinguishable pair (p,q) or (q,p) in D.
$Q(A_r) := \{[p] \mid \exists q \in Q(A) : od(p,q) \in D\}$;
$q_{A_r} := [q_A]$;
$\delta_{A_r} := \{([p],x) \mapsto ([\lambda_A(p,x)], \omega_A(p,x)) \mid (p,x) \in Q_A \times I\}$;
// Well-definedness of δ_{A_r} follows from properties of
// equivalence classes $[p]$.
$A := A_r$;
end

Strukturelles Testen

Strukturelle Tests können auf unterschiedlicher Grundlage durchgeführt werden:

- **Test gegen Anforderungsmodell:** Strukturelle Tests gegen das Anforderungsmodell zielen auf die Abdeckung bestimmter "Regionen" des Spezifikationsmodells ab, z. B.
  - Zustände
  - Transitionen und daran geknüpfte Bedingungen
  - Ereignisfolgen

Strukturelles Testen

- **Test gegen Architekturmodell:**
  - Verwendung bestimmter Schnittstellen
  - Verwendungt bestimmter Systemkomponenten (Rechner, Busse, ...)
  - Stimulation bestimmter Prozesse/Threads/Methoden/Funktionen

in bestimmter Reihenfolge.

Strukturelles Testen

- **Test gegen Code-Kontrollstrukturen ("Codeüberdeckungstests"):** Ausführung bestimmter
  - Anweisungen
  - zusammengesetzter Bedingungen
  - atomarer Bedingungen

in bestimmter Reihenfolge. Codeüberdeckungstests werden auch **White-Box Tests** genannt, da sie die Offenlegung des Programmcodes erfordern.

Strukturelles Testen

Bei strukturellen Tests gegen Architekturmodell und Code-Kontrollstrukturen muss das beobachtete Verhalten des SUT gegen die Anforderungsspezifikation geprüft werden!

Strukturelles Testen

- Codeüberdeckungstests werden auch **White-Box Tests** genannt, da sie die Offenlegung des Programmcodes erfordern.
- **Bei strukturellen Tests gegen Architekturmodell und Code-Kontrollstrukturen muss das beobachtete Verhalten des SUT gegen die Anforderungsspezifikation geprüft werden!**
- Allgemeiner: Strukturelle Tests bestimmen die **Testfall-/Testdatenauswahl** – die **Testauswertung** erfolgt gegen eine Anforderungsspezifikation.

Strukturelles Testen

Bei strukturellen Tests gegen Architekturmodell und Code-Kontrollstrukturen muss das beobachtete Verhalten des SUT gegen die Anforderungsspezifikation geprüft werden!

Strukturelles Testen – Codeüberdeckungstests

Die wichtigsten Varianten der Codeüberdeckungstests sind

- Anweisungsüberdeckung (Statement Coverage)
- Verzweigungsüberdeckung (Decision Coverage, Branch Coverage)
- Minimale Mehrfachbedingungsüberdeckung (Modified Condition/Decision Coverage, MC/DC)
- Pfadüberdeckung (Path Coverage)

Strukturelles Testen – Codeüberdeckungstests

Anweisungsüberdeckung fordert die mindestens einmalige Ausführung jeder Anweisung im Rahmen der gesamten Testsuite.

Verzweigungsüberdeckung fordert

1. die mindestens einmalige Ausführung jeder Anweisung im Rahmen der gesamten Testsuite,
2. die mindestens einmalige Auswertung jeder Bedingung mit `true`,
3. die mindestens einmalige Auswertung jeder Bedingung mit `false`.

Strukturelles Testen – Codeüberdeckungstests

Bedingung 1. sichert ab, dass Verzweigungsüberdeckung auch Anweisungsüberdeckung impliziert:

Unbedingte Sprünge (`return, goto`) können lineare Anweisungsfolgen überspringen. Man kann also Bedingungen 2. und 3. vollständig erfüllen, ohne dass alle linearen Anweisungsfolgen überdeckt sind.

Strukturelles Testen – Codeüberdeckungstests

Minimale Mehrfachbedingungsüberdeckung (MC/DC Coverage) verfeinern die Testdatenauswahl in Verzweigungsbedingungen, und damit die "normale" Verzweigungsüberdeckung:

- Für eine if-Bedingung `if ( a and b ) { ...}` sind mindestens folgende Fälle zu überprüfen, insofern sie realisierbar sind:
  - `a == true, b == true`
  - `a == false, b == true`
  - `a == true, b == false`

Strukturelles Testen – Codeüberdeckungstests

- Für eine if-Bedingung `if ( a or b ) { ...}` sind mindestens folgende Fälle zu überprüfen, insofern sie realisierbar sind:
  - `a == true, b == false`
  - `a == false, b == true`
  - `a == false, b == false`

Motivation für die MC/DC Definition: Für jede Bedingung sollen alle minimalen Literalvaluationen geprüft werden, die für das Auswertungsergebnis `true` oder `false` verantwortlich sind. Auf diese Weise lassen sich **stick-at-zero** und `stuck-at-one` Fehler aufdecken.

Strukturelles Testen – Codeüberdeckungstests

Nicht jede geforderte Kombination ist immer realisierbar: beispielsweise für die Bedingung

```
unsigned int x,y,z,n;
if ( n <= 2 and x**n + y**n = z**n ) { ... }
```

können wir nur die Kombinationen

`n <= 2`	`x**n + y**n = z**n`
`true`	`true`
`true`	`false`
`false`	`false`

testen; einen Fall `n > 2 and x**n + y**n = z**n` gibt es nicht (folgt aus Fermats letztem Theorem).

Strukturelles Testen – Codeüberdeckungstests

Verallgemeinerte MC/DC-Bedingungen für beliebige logische Ausdrücke: Die logische Bedingung wird wahlweise in **konjunktive oder disjunktive Normalform gebracht**. Wir arbeiten im folgenden mit der CNF, so dass sich die Bedingung durch

$$\bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{m_i} x_{ij} \right)$$

Dabei sind die x_{ij}, d. h. atomare Boolsche Ausdrücke, die eventuell negiert sein können (negative Literale).

Strukturelles Testen – Codeüberdeckungstests

Die minimalen Literalvaluationen, die die Bedingung zu `true` auswerten lassen, sind durch folgende Funktion zu konstruieren:

$$\Phi(i, j, k_1, \ldots, k_n) = \begin{cases} 1 & j = k_i \\ 0 & \text{sonst} \end{cases}$$

ist definiert auf

$$\{(i, j, k_1, \ldots, k_n) \mid i \in \{1, \ldots, n\}, j \in \{1, \ldots, m_i\}, k_i \in \{1, \ldots, m_i\}\}$$

Für MC/DC-Überdeckung sind jetzt folgende über $k_1, \ldots, k_n$ indizierte Literalvaluationen zu testen, soweit möglich:

$$x_{ij}^{k_1,\ldots,k_n} = \Phi(i, j, k_1, \ldots, k_n)$$

Es gibt also maximal $m_1 \cdot m_2 \cdots m_n$ zu testende Kombinat

Strukturelles Testen – Codeüberdeckungstests

Die minimalen Literalvaluationen, die die Bedingung zu `false` auswerten lassen, sind durch folgende Funktion zu konstruieren:

$$\Psi(i, j, i_0, k_1, \ldots, k_n) = \begin{cases} 0 & i = i_0 \\ 1 & i \neq i_0 \wedge j = k_i \\ 0 & \text{sonst} \end{cases}$$

ist definiert auf

$$\{(i, j, i_0, k_1, \ldots, k_n) \mid i, i_0 \in \{1, \ldots, n\}, j \in \{1, \ldots, m_i\}, k_i \in \{1, \ldots, m_i\}\}$$

Äquivalenzklassentests

Für MC/DC-Überdeckung sind jetzt folgende über $i_0, k_1, \ldots, k_n$ indizierte Literalvaluationen zu testen, soweit möglich:

$$x_{ij}^{i_0,k_1,\ldots,k_n} = \Psi(i, j, i_0, k_1, \ldots, k_n)$$

Es gibt also maximal

$$\Sigma_{i=0}^{n}(m_1 \cdots \hat{m}_i \cdots m_n)$$

zu testende Kombinationen, wobei $\hat{m}_i$ bedeutet, dass dieser Term im Produkt durch den Wert 1 ersetzt wird.

Äquivalenzklassentests

Äquivalenzklassentests dienen der Einteilung des Eingaberaums in Klassen, von denen erwartet werden kann, dass der Testling sie auf "äquivalente" Weise verarbeitet.

Äquivalente Verarbeitung bedeutet dabei: Jedem durch einen Repräsentanten der Klasse erzeugten Verarbeitungspfad werden

- dieselben Kontrollentscheidungen gefällt, und daher
- dieselben Datentransformationsanweisungen durchgeführt.

Äquivalenzklassentests

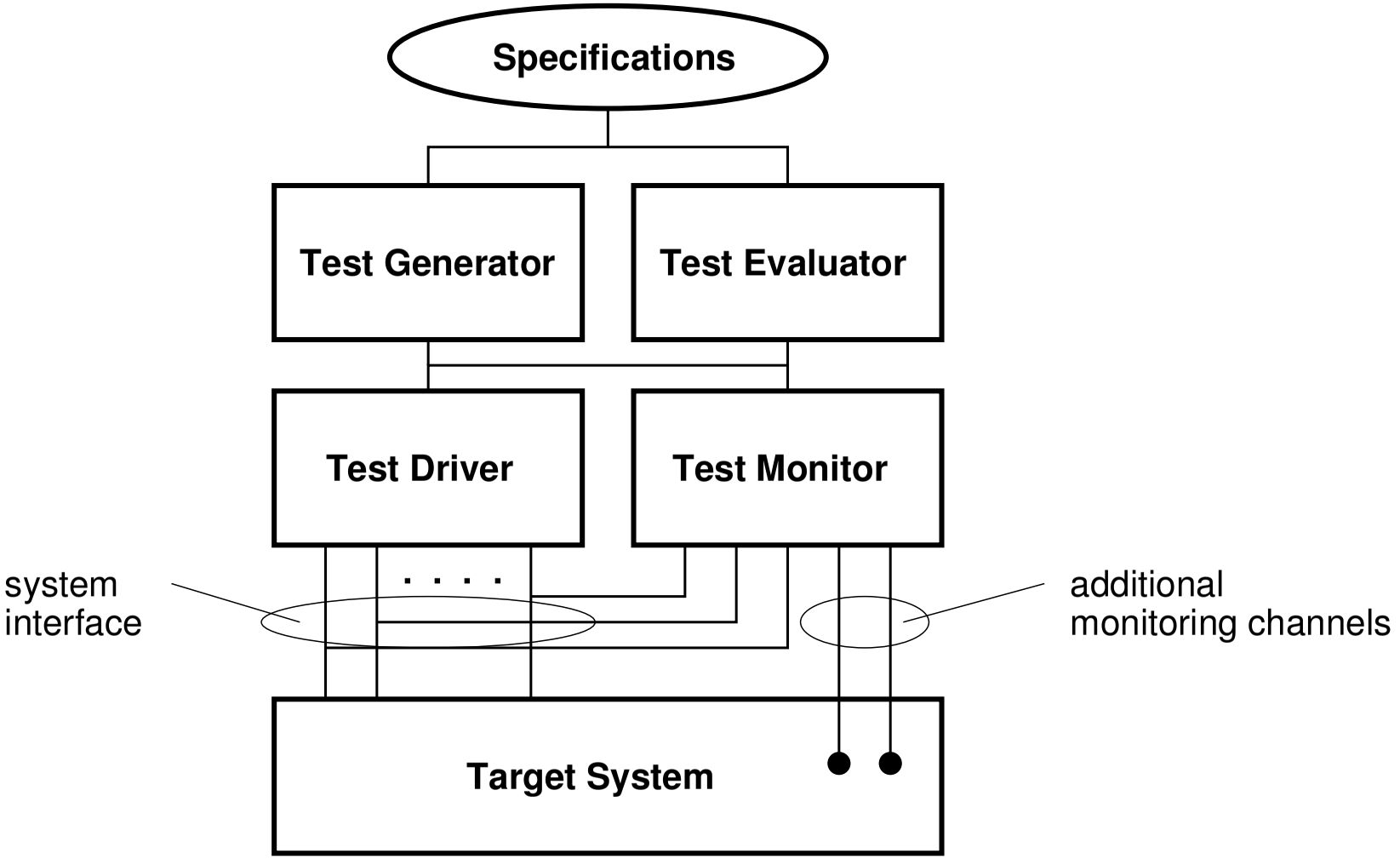
Aus diesem Grund lässt sich mit einer gewissen Wahrscheinlichkeit annehmen, dass viele Fehler, die auf einem Verarbeitungspfad des SUT liegen, durch nahezu jeden Repräsentanten der Klasse aufgedeckt werden können. Daher gilt es als hinreichend, aus jeder Klasse nur wenige Repräsentanten zu erproben (siehe Thema "Grenzwerttests" unten).

Umgekehrt sollte versucht werden, aus jeder Äquivalenzklasse A_1 mindestens einen Repräsentanten $a \in A_1$ zu testen, da Repräsentanten anderer Klassen A_2 niemals dieselbe Anweisungsfolge im SUT stimulieren werden, welche a_1 erzeugt.

Äquivalenzklassentests

- Äquivalenzklassen werden in der Regel zuerst als Black-Box Funktionstests entworfen.
- Liegt ein Funktionsmodell vor, können die Klassen anhand der möglichen Pfade durch das Modell identifiziert werden.
- Durch Analyse der beim funktionsbasierten Äquivalenzklassentest erzielten Pfadüberdeckung des SUT Codes stellt man fest, ob bestehende Äquivalensklassen noch weiter verfeinert werden müssen.
- Die im SUT-Code zu überdeckenden Pfade identifziert man am besten anhand des **Kontrollflussgraphen** des Testlings.

Tools und Auswahlkriterien



Design for Testability – Kompositionalität

- **Kompositionalität:** Komponenten erfüllen ihre Spezifikation unabhängig davon, ob weitere Komponenten parallel aktiv sind oder nicht.
  - **Beispiel "nicht kompositionell":** zwei Prozesse greifen unsynchronisiert auf globale Variable zu
  - **Beispiel "kompositionell":** zwei Prozesse greifen mittels Monitor auf globale Variable zu

Design for Testability – Kompositionalität

Auswirkung der Kompositionalität auf das Testen:

- Funktionstests können mit den isolierten Komponenten ausgeführt werden.
- Integrationstests müssen nur noch die **korrekte Kooperation** prüfen (Strukturtest).

Design for Testability – Kompositionalität

Herstellung von Kompositionalität im Design – Kompositionalität bzgl. Datentransformation und Kausalität:

- Zugriff auf globale Daten kapseln
- Zugriff auf kritische Abschnitte in gleichförmiger Art synchronisieren:
  - Verwendung von Monitoren
  - Verwendung von Semaphoren
  - Verwendung von Active-Wait-Mutex Verfahren

Achtung: bei Kompositionalität bzgl. Timing-Eigenschaften müssen ggf. parallele Komponenten auf mehrere Prozessoren verteilt werden!

Design for Testability – Strukturerhaltende Transformation von Anforderungen ins Design

- **Strukturerhaltende Transformation:** Struktur und Datenfluss der Anforderungsspezifikation wird für das Architekturdesign übernommen $\Longrightarrow$ direkte Beziehung zwischen Systemkomponente und der von ihr implementierten funktionalen Anforderung.

Vorteil der strukturerhaltenden Transformation beim Testen:

- **Überdeckung der funktionalen Anforderungen beim Testen erzeugt gleichzeitig bereits die Strukturüberdeckung.**

Design for Testability – HW/SW Testschnittstellen

Zielsetzung: Bereits im Design "Testpunkte", d.h. zusätzliche Schnittstellen für den späteren Test einplanen – am besten als permanente Systemschnittstellen vorsehen. Anwendungsbereiche:

- Monitoring der erzielten Überdeckung
- Künstliche Herbeiführung von im operationellen Betrieb selten auftretenden Zuständen
- Fehlerinjektion auf Software und Hardwareebene

Design for Testability – Kohäsion

- **Kohäsion:** SW-Komponenten tragen zur Implementierung von nur einer funktionalen Anforderung bei – es werden nicht mehrere funktionalen Anforderungen gleichzeitig in derselben Komponente bearbeitet.

Vorteile beim Testen:

- Reduktion der Anzahl erforderlicher Testfälle

Design for Testability – Separation von Kontrolle und Datentransformation

- **Separation von Kontrolle und Datentransformation:**
  - Kontrolle wird von übergeordneten Komponenten ausgeübt, die keine Datentransformationen durchführen.
  - Datentransformationen werden von untergeordneten Komponenten ausgeführt, die möglichst linear arbeiten und von den Kontrollkomponenten aktiviert werden.

Design for Testability – Separation von Kontrolle und Datentransformation

Vorteile beim Testen:

- Für den Test der Kontrollkomponenten kann von Nutzdaten weitgehend abstrahiert werden.
- Für die Erzielung der erforderlichen (Verzweigungs-)Überdeckung bei Datentransformationen sind weniger Testfälle erforderlich.

Design for Testability – Kapselung der kritischen Funktionen, Partitionierung

- **Partitionierung:** Trennung kritischer Funktionen von unkritischen Funktionen im Systemdesign – Ausführung der unkritischen Funktionen muss nebenwirkungsfrei für die kritischen Funktionen sein.
- **Kapselung:** Implementierung der kritischen Funktionen in möglichst wenigen Komponenten.

Design for Testability – Kapselung der kritischen Funktionen, Partitionierung

Aus Partitionierung und Kapselung resultiert verminderter Testaufwand:

- Unkritische Funktionen können mit geringerer Überdeckung geprüft werden. ($\Longrightarrow$ Das gilt nur bei guter Partitionierung – sonst kann ein Dominoeffekt von unkritischen zu kritischen Fehlern erzeugt werden!)
- Nur wenige Funktionen sind mit maximaler Überdeckung zu prüfen.
- Kein Integrationstest für kritische Funktionen im Zusammenspiel mit unkritischen erforderlich.

Design for Testability – Probleme beim OO-Design

Folgende OO-Merkmale erschweren den vertrauenswürdigen Test von OO-Komponenten:

- **Dynamische Objekterzeugung/Objektvernichtung:** Das korrekte Verhalten hängt nicht allein von der Programmierung der Objekte, sondern von den Zeitpunkten der Objektallokation/-deallokation ab. (**Analoges Problem:** Polymorphie und Vererbung mit dynamischer Bindung)

- **Generische Klassen:** Die Korrektheit des Codes für die generische Klasse garantiert noch nicht die Korrektheit jeder möglichen Instanz, da die konkreten Datentypen der Instantiierung besondere Probleme erzeugen können, die auf Klassenebene nicht sichtbar sind.

Anhang: Begriffe

Analytische Qualitätssicherung Sicherung der Qualität durch Untersuchung der Produkteigenschaften

Prozessbezogene Qualitätssicherung Sicherung der Qualität durch Prüfung der Angemessenheit und korrekten Ausführung des Entwicklungsprozesses

Test Case (RTCA/DO-178B) A set of inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

Testen analytische Qualitätssicherung von ausführbaren SW-Programmen oder vollständigen HW/SW-Systemen

Testing (RTCA/DO-178B) The process of exercising a system or system component to verify that it satisfies specified requirements and to detect errors.

Test Procedure (RTCA/DO-178B) Detailed instructions for the set-up and execution of a given set of test cases, and instructions for the evaluation of results of executing the test cases.

Dynamisches Testen Prüfung von ausführbaren SW-Programmen oder vollständigen HW/SW-Systemen durch dynamische Ausführung mit speziellen Eingabedaten

Statische Analyse analytische Qualitätssicherung von Software durch Untersuchung des Quellcodes ohne seine dynamische Ausführung

Validation Prüfung des Systems (oder seiner Anforderungsspezifikation) hinsichtlich Wirksamkeit (Adäquatheit), Vollständigkeit, Widerspruchsfreiheit, Eindeutigkeit

Validation (RTCA/DO-178B) The process of determining that the requirements are correct requirements and that they are complete. The sysrem life cycle process may use software requirements and derived requirements in system validation.

Verifikation Prüfung eines Produktes gegen eine Referenzspezifikation

Verification (RTCA/DO-178B) The evaluation of the results of a process to ensure correctness and consistency with respect to the inputs and standards provided to that process.

Formale Verifikation Prüfung eines Produktes gegen eine formale Spezifikation mittels mathematischer Nachweisverfahren