
Kap. 7:

Entwicklung von Anwendungen

(c) Validierung & Test

7. Entwicklung von Anwendungen

Konzentration auf

- Sicherheitskritische Anwendungen
- Echtzeitanwendungen

Gliederung

(a)

1. Einführung
2. IEC EN 61508
3. Programmiersprachen (MISRA C/C++, Ada)

(b)

4. Modellierung (SysML, UML MARTE, ...)

(c)

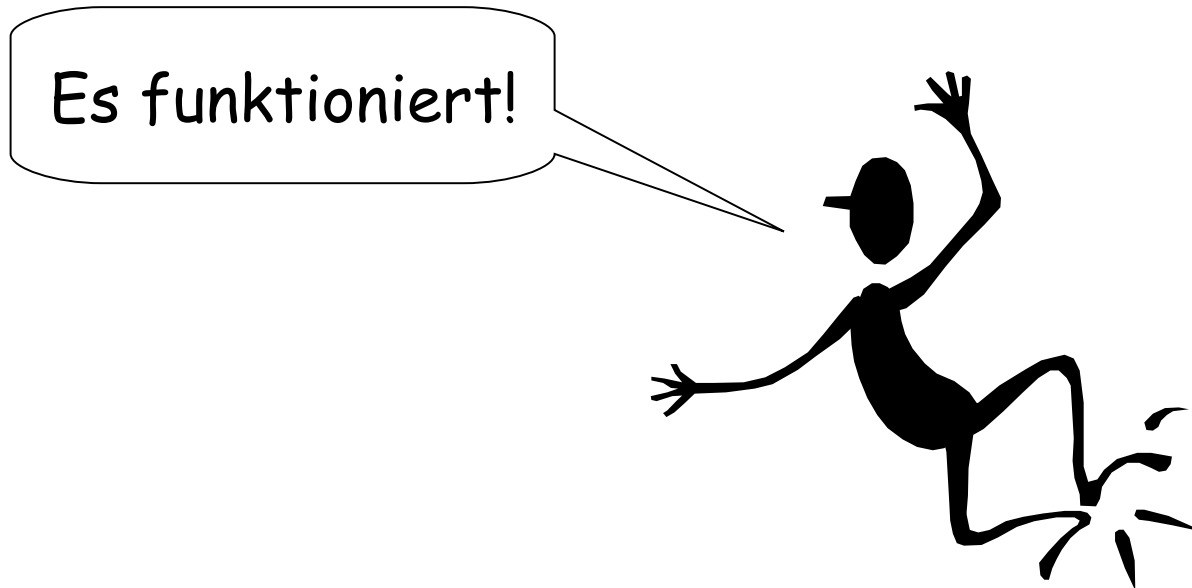
5. Validierung
6. Systematisches Testen

7.5 Validierung

- **Häufiger Einsatz von Echtzeitsoftware in sicherheitskritischen Bereichen (z.B. ABS, Avionik, ..)**
- **Je nach Einsatzgebiet existieren Vorschriften für die Zertifizierung von Software**
- **z.B.:**
 - **DO-178A (FAA)**
 - **IEC 61508**

7.6 Systematischen Testen

- Ad-hoc Methoden...



- ... sind unzureichend!
 - Eher „Debugging/Inbetriebnahme“ als „Test“
 - Qualität ist nicht nachvollziehbar
 - Entwickler sind schlechte Tester (ihrer eigenen Programme)

<http://www.fbe.hs-bremen.de/spillner/systtest>

- **Ziel von Tests (bzw. der Validierung):**
 - Nachweis der Existenz von Fehlern in einem Programm.
 - Erfolgserlebnis des Testers: „Fehler gefunden“.
- **Ziel der Verifikation:**
 - Nachweis der nicht-Existenz von Fehlern in einem Programm.
 - Erfolgserlebnis des Verifizierens: „Beweis der Funktion“.
- **Problem bei beiden: Codekomplexität**
 - Hoher Aufwand bei Validierung
 - Verifikation komplexer Gewerke (heute noch) unmöglich

- **Geordnete Abfolge von Testfällen**
- **muss jederzeit reproduzierbare Ergebnisse liefern**
- **automatisierter Ablauf (keine Benutzerinteraktion)**
- **modular aufgebaut -> erweiterbar**
- **z.B. shell-Skript, spezielles Anwenderprogramm, vordefinierte Abfolge von Benutzereingaben,**
- **Vorzugsweise realisiert mit Hilfe eines „Test Harness“ (z.B. DejaGNU)**

- **Testen auf Rückschritte (engl. regression)**
- **Es kommt immer wieder zu Rückschritten im Zuge von Weiterentwicklung/Pflege:**
 - Fehler bei Hinzufügen neuer Funktionalität
 - „verschlimmbessern“
 - Fehler im Umgang mit Revisionskontrollwerkzeugen
 - Beheben von Symptomen anstelle der Ursachen
 - Schlechte Kommentierpraxis im Team
- **Testsuite als Bestandteil des Produkts (Weiterpflege):**
 - Neue Funktionen im Produkt: neue Testfälle
 - Fehler im Produkt:
 - 1. Neuer Testfall, der den Fehler reproduziert
 - 2. Fehler beheben
 - 3. Testfall verbleibt in der Testsuite

- **Unit Test**
 - Schwierigkeit/Unmöglichkeit, bei komplexen Gewerken Programmteile in tieferen Schichten über die Programmierschnittstelle gezielt anzusprechen
 - Aufteilen des Gewerkes in kleinere Module (*Units*)
 - evtl. emulieren der externen Schnittstellen durch „stubs“
 - Test der einzelnen Units
- **Integrationstest**
 - Gemeinsames Testen der Units im Zusammenspiel
 - Haupt-Augenmerk dabei auf Schnittstellen
- **Systemtest**
 - Testen des gesamten Systems

- **Schrittweises Entwickeln von Testfällen ausschließlich anhand der Funktionsbeschreibung („*Black-Box*“-Test)**
- **Eigenschaften werden Punkt für Punkt (in Form einzelner Testfälle) überprüft**

```
READ(2)                                UNIX Programmer's Manual
READ(2)

NAME
    read - read from a file descriptor

SYNOPSIS
    #include <unistd.h>

    ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
    read() attempts to read up to count bytes from file descriptor fd into
    the buffer starting at buf. (...)

RETURN VALUE
    On success, the number of bytes read is returned (...).
    On error, -1 is returned, and errno is set appropriately. (...)

ERRORS
    EBADF  fd is not a valid file descriptor or is not open for
    reading.

    EFAULT buf is outside your accessible address space.
```

- **Vorgehensweise:**
 - Dokumentierter, umkehrbar eindeutiger Zusammenhang von Eigenschaften und Testfällen

<p>READ(2) UNIX Programmer's Manual READ(2)</p> <p>NAME</p> <p>read - read from a file descriptor</p> <p>SYNOPSIS</p> <pre>#include <unistd.h></pre> <pre>ssize_t read(int fd, void *buf, size_t count);</pre> <p>DESCRIPTION</p> <p>read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf. (...)</p> <p>RETURN VALUE</p> <p>On success, the number of bytes read is returned (...)</p> <p>On error, -1 is returned, and errno is set appropriately. (...)</p> <p>ERRORS</p> <p>EBADF fd is not a valid file descriptor or is not open for reading.</p> <p>EFAULT buf is outside your accessible address space.</p>	<p>test_read_EBADF()</p> <pre>{ int badfd = 1234567; char buf[SIZE]; if(read(badfd, buf, SIZE) != -1) { printf("Testcase failed (1)\n"); return ERROR; } else if(errno != EBADF) { printf("Testcase failed (2)\n"); return ERROR; } return OK; }</pre> <p>test_read_EFAULT()</p> <pre>{ int fd =; if(read(fd, NULL, 1) != -1) { printf("Testcase failed (3)\n"); return ERROR; } else if(errno != EFAULT) { printf("Testcase failed (4)\n"); return ERROR; } return OK; }</pre>
---	--

- **Stärken**
 - Führt bei vollständiger Spezifikation zum vollständigen Test (theoretisch)
 - Keine Abhängigkeit von der Implementierung
 - Daher anwendbar auf alle Implementierungen der gleichen Schnittstelle
- **Schwächen**
 - Qualität des Ergebnisses steht und fällt mit der Vollständigkeit der Funktionsbeschreibung
 - Liefert keine Hinweise auf Lücken in der Spezifikation
 - „Toter“ Code wird weder erkannt noch getestet
 - Vollständiger Test i.A. nicht praktikabel

- **vollständiger Test i.A. nicht praktikabel**

```
int function(int x)
{
    switch(x) {
        case 1:
            ....
        case 135:
            ....
    }
```

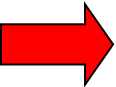


2^{32} Testfälle!

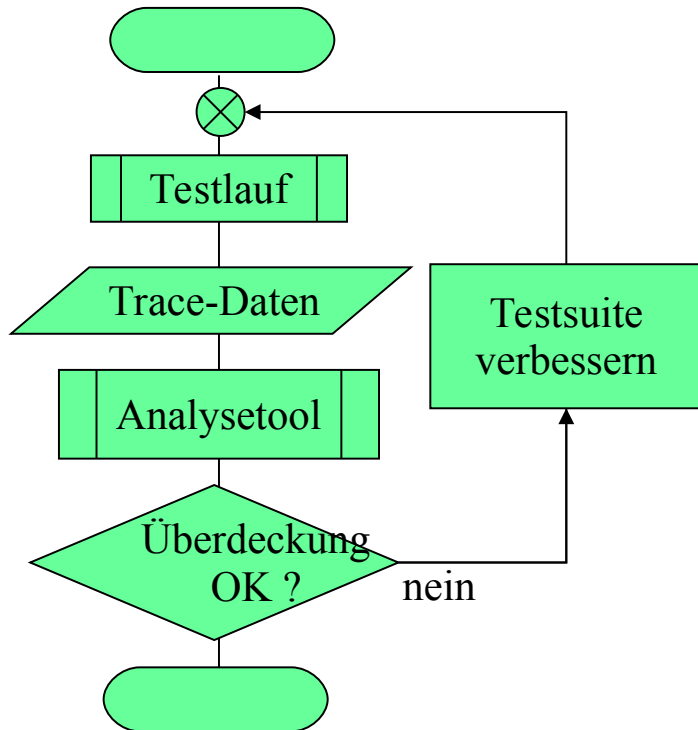
- I.A. ist der bei weitem größte Teil der Testfälle redundant
(=> „Äquivalenzklassen“)
- Reduktion auf relevante Testfälle erfordert Einbeziehung des Quellcodes.

- auch: „White-Box“ / „Glass-Box“-Test)
 - Ausgangsbasis: Quellcode des Prüflings
 - Ziel: Alle Programmteile des Prüflings sollen berührt (*überdeckt*) werden

<pre>unsigned char toupper(unsigned char c) { /* Note: this also converts international character codes(0xc0 .. 0xfe) */ if((c >= 'a' && c <= 'z') (c >= 0xc0 && c <= 0xfe)) { return(c - 'a' + 'A'); } else { return c; } }</pre>	<pre>struct testcase { const unsigned char in, expect; }; static struct testcase tc[4] = { {'a', 'A', }, {0xe0, 0xc0, }, {'A', 'A', }, {0xff, 0xff, } }; test_toupper() { int i; for(i = 0; i <= 4; i++) { if(tc[i].expect != toupper(tc[i].in)) { printf("Test %d failed\n", i); } } }</pre>
---	--

 **4 Testfälle
(statt 256)**

- **Vorgehensweise**



- Instrumentierung mit Hilfe eines Überdeckungswerkzeuges (Coverage Tool)
- Instrumentiertes Programm verhält sich (idealerweise) wie original-Programm, erzeugt aber zusätzlich Trace-Daten
- Analysewerkzeug ermittelt anhand der Trace-Daten Überdeckungsmaße und „weiße Stellen“
- Erweiterung der Testsuite zur Verbesserung der Überdeckung
- Iterativ optimieren, bis geforderte Überdeckungsmaße erreicht werden

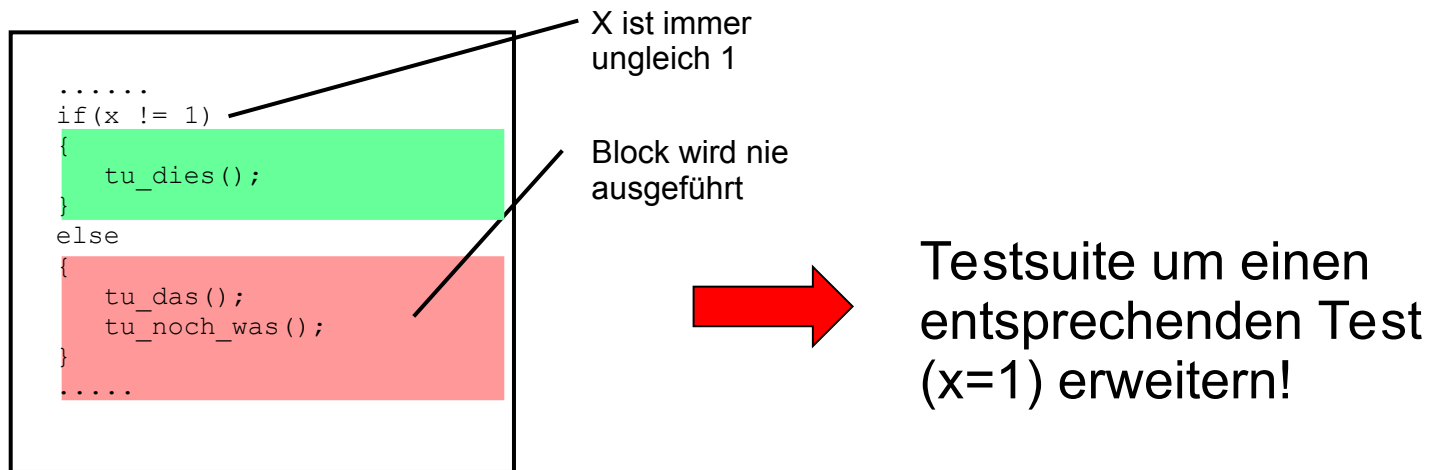
- **Stärken**
 - Erkennt redundante Testfälle
 - Erkennt „toten Code“
 - Quantitative Erfassung der Testtiefe (?)
- **Schwächen**
 - Test ist spezifisch für die Implementierung
 - Keine Hinweise auf Unterlassungsfehler:

```
int divide(int x, int y)
{
    return (x / y);
}
```

Fehler:
Prüfung auf $y \neq 0$ fehlt!

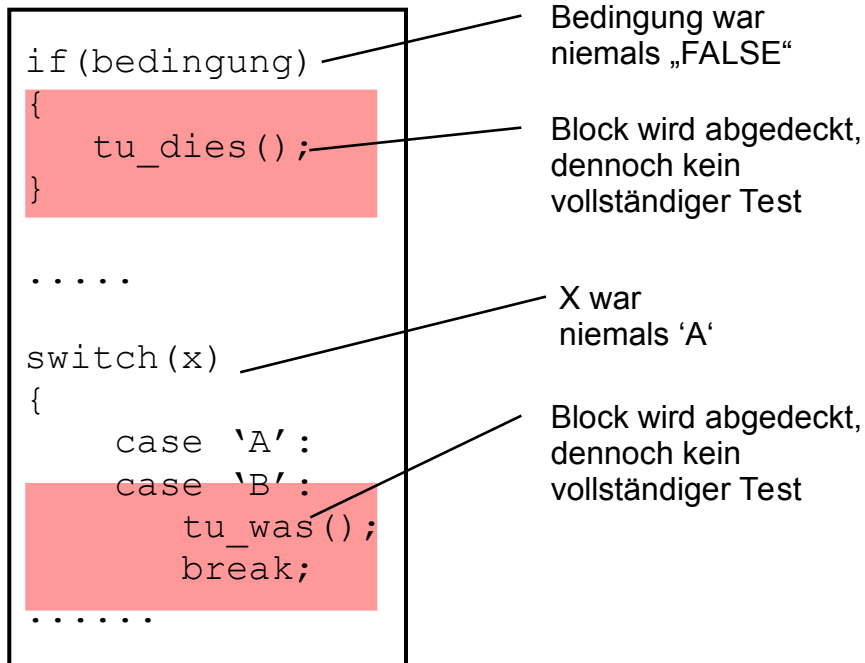
Struktureller Test: Überdeckungsmaße (1) 7.6

- **Eintrittsüberdeckung (function/entry coverage)**
 - Wurden alle Funktionen aufgerufen ?
- **Anweisungsüberdeckung (statement coverage)**
 - Wurde sämtlicher Code des Prüflings mindestens einmal ausgeführt ?
- **Blocküberdeckung (block coverage)**
 - Wurden sämtliche basic blocks ausgeführt ?

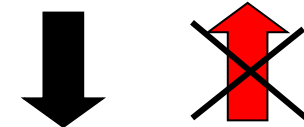


Struktureller Test: Überdeckungsmaße (2) 7.6

- Zweigüberdeckung(decision/branch coverage)
 - Sind alle möglichen Alternativen aller Bedingungen aufgetreten ?



100% Zweigüberdeckung



100% Anweisungsüberdeckung

Struktureller Test: Überdeckungsmaße (3) 7.6

- **Mehrfach-Bedingungsüberdeckung (multiple condition coverage)**
 - Sind alle möglichen Kombinationen aller Bedingungen aufgetreten ?

```
if (a && (b || c))  
{  
    tu_was();  
}
```

	a	b	c	Ergebnis
Komb. 1	FALSE	FALSE	FALSE	FALSE
Komb. 2	FALSE	FALSE	TRUE	FALSE
Komb. 3	FALSE	TRUE	FALSE	FALSE
Komb. 4	FALSE	TRUE	TRUE	FALSE
Komb. 5	TRUE	FALSE	FALSE	FALSE
Komb. 6	TRUE	FALSE	TRUE	TRUE
Komb. 7	TRUE	TRUE	FALSE	TRUE
Komb. 8	TRUE	TRUE	TRUE	TRUE

N Bedingungen -> 2^N Kombinationen

Bestimmte Kombinationen aufgrund von Shortcuts u.U. nicht erreichbar

Struktureller Test: Überdeckungsmaße (4) 7.6

- **Modifizierte Bedingungsüberdeckung (modified condition/decision coverage, MC/DC)**
 - **Reduktion auf diejenigen Kombinationen, bei denen die Änderung einer Bedingung das Ergebnis beeinflusst**

```
if (a && (b || c))  
{  
    tu_was();  
}
```

	a	b	c	Ergebnis
Komb. 1	FALSE	FALSE	FALSE	FALSE
Komb. 2	FALSE	FALSE	TRUE	FALSE
Komb. 3	FALSE	TRUE	FALSE	FALSE
Komb. 4	FALSE	TRUE	TRUE	FALSE
Komb. 5	TRUE	FALSE	FALSE	FALSE
Komb. 6	TRUE	FALSE	TRUE	TRUE
Komb. 7	TRUE	TRUE	FALSE	TRUE
Komb. 8	TRUE	TRUE	TRUE	TRUE

N Bedingungen -> N+1 <= Anzahl Kombinationen <= 2*N

Relevant bei Anwendungen in der Avionik (DO-178B)

- **Pfadüberdeckung („path coverage“)**
 - Sind alle möglichen Pfade ausgeführt worden ?

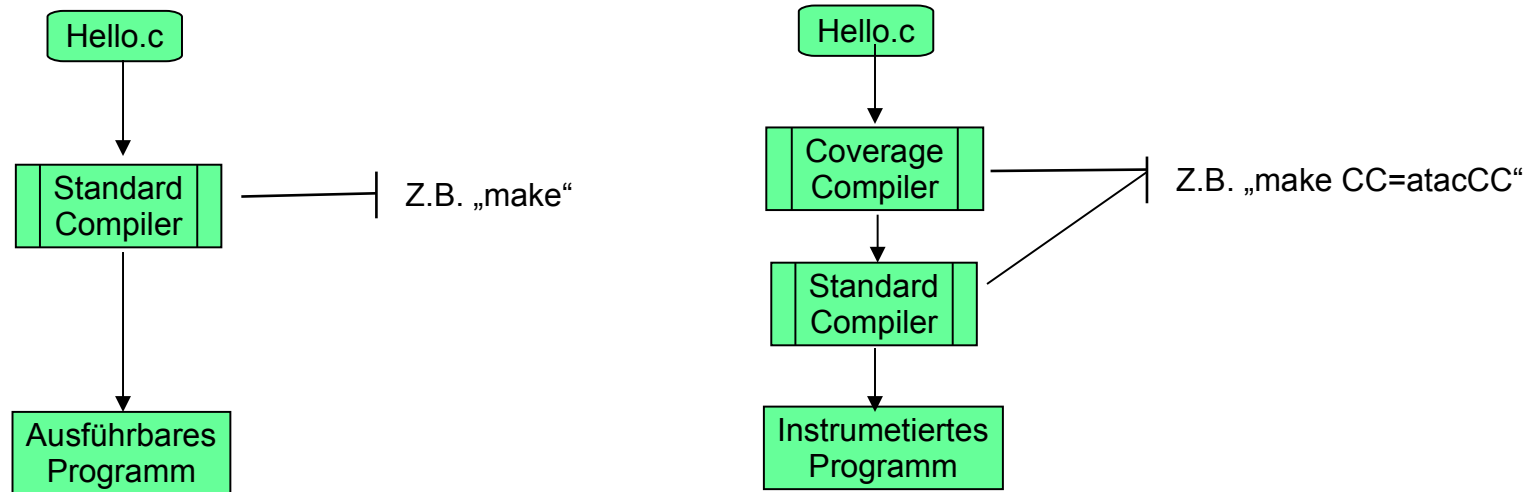
```
if (bedingung1)
    tu_dies();
if (bedingung2)
    tu_das();
if (bedingung3)
    tu_nochwas();
```

	Bedingung1	Bedingung2	Bedingung3
Pfad1	FALSE	FALSE	FALSE
Pfad2	FALSE	FALSE	TRUE
Pfad3	FALSE	TRUE	FALSE
Pfad4	FALSE	TRUE	TRUE
Pfad5	TRUE	FALSE	FALSE
Pfad6	TRUE	FALSE	TRUE
Pfad7	TRUE	TRUE	FALSE
Pfad8	TRUE	TRUE	TRUE

**Aufwand steigt exponentiell mit der Anzahl der Bedingungen
-> i.A. zu Komplex !**

- Defs/Uses Kriterium (DU-Path)
 - Reduzierte Variante der Pfadüberdeckung
 - Es wird nur die Untermenge der Pfade zwischen der Wertzuweisung einer Variablen und der Referenz auf den Wert der Variablen betrachtet
 - Computational use coverage („C-use“): Referenz auf die Variable ist nicht Teil einer Bedingung
 - Predicate use coverage („P-use“): Referenz ist Teil einer Bedingung -> alle Verzweigungen aufgrund des Wertes der Bedingung werden zu der Liste betrachteten Pfade gezählt

- **Vorgehensweise:**
 - Das Coverage Tool liest Quellcode (z.B. „C“) und erzeugt „instrumentierten“ Quellcode,
 - Der instrumentierte Code wird mit Hilfe des Standard-Compilers in Binärcode übersetzt
 - Oft transparenter Vorgang (Substitution des Standard-Compilers).



- **Beispiel: „ATAC“ (rein softwarebasiert)**
 - „Automatic Test Aalysis for C Programs“ (Horgan/London '91)
 - Es gibt auch hardwaregestützte Werkzeuge
 - -> geringere Verfälschung des Laufzeitverhaltens

Originalcode:

```
main()
{
    printf("Hello World\n");
}
```

Instrumentierter Code:

```
main()
{
    struct Ztables *ZTpnr = &ZTmain;
    void *ZBfr = alloca(
        sizeof(struct Zcontext) +
        ZTpnr->max_puse * sizeof(struct Zpdef)
        + ZTpnr->nvar * sizeof(short));
    int Z=aTaC(0, (int) &ZTmain, ZTpnr, ZBfr);
    {
        aTaC(Z, 1, ZTpnr, ZBfr);
        printf("Hello World\n");
    }
}
```

Funktion aus Laufzeitbibliothek

- **Vorgehensweise**
 - **Das instrumentierte Programm verhält sich (idealerweise) genauso wie das nicht-instrumentierte Programm**
 - **Es erzeugt zusätzlich zur Laufzeit „trace“-Daten**
 - **Auswertung der trace-Daten mit Analysetools liefert z.B.**
 - » **Block Coverage (Prozent/absolut)**
 - » **Decision Coverage (Prozent/absolut)**
 - » **C-uses/P-uses (Prozent/absolut)**
 - » **Quellcode-Listing mit Markierung der nicht-abgedeckten Fälle**

- **Besonderheiten bei embedded Systems**
 - höherer Ressourcenbedarf durch Instrumentierung
 - Probleme durch Verfälschung des Zeitverhaltens
 - evtl. kein Massenspeicher für Trace-Daten
- **Mögliche Abhilfen**
 - Ausweichen auf Testrechner mit ausreichend Ressourcen
 - Hardwaregestützte Werkzeuge

- **Instrumentieren von Betriebssystemcode**
 - Keine Standard-C Bibliothek
 - BS-Code verwendet häufig non-Standard Konstrukte (assembler-inline, pragma, ..), die vom Instrumentierer (=eigenständiger Compiler) u.U. nicht unterstützt werden
 - Fehlender Dateisystemzugriff für Trace-Daten
- **Mögliche Abhilfen**
 - Implementierung der fehlenden Funktionen als Pseudotreiber (erfordert Quellcode)
 - Hardwaregestützte Werkzeuge
 - Evtl. Instrumentierung lokal unterbinden

Vergleich: Funktionaler vs. Struktureller Test 7.6

- **Rein funktionales Testen *kann* sinnvoll sein**
 - Unabhängigkeit der Testsuite von der Implementierung des Prüflings
 - z.B. Test auf Standard API Konformität
- **Rein strukturelles Testen ist niemals sinnvoll**
 - 100%ige Überdeckung trotz schwerster (Unterlassungs-) Fehler möglich
 - » („A fool with a tool is still a fool“)
- **Struktureller Test als Ergänzung zum funktionalen Test**
 - 1) Entwickeln einer funktionalen Testsuite
 - 2) Optimierung der Testsuite anhand strukturellem Test