

Phase 3: Komponenten (Testen)

Echtzeitssystemelabor - Vorlesung/Übung

**Peter Ulbrich
Wolfgang Schröder-Preikschat**

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander Universität Erlangen-Nürnberg

<http://www4.cs.fau.de/~{ulbrich,wosch}>
{ulbrich,wosch}@cs.fau.de



Übersicht

- Warum Testen?
- Testarten
- Wo kommen Testfälle her?
- Wie gut hat man getestet?
- Spezifikation von Testfällen
- Implementierung von Testfällen
- Performanztests



Warum Testen?

- verschiedene Möglichkeiten, Aussagen über Programme zu treffen:
 - **informelle Methoden**
 - Inspection, Review, Walkthrough, ...
 - **analytische Methoden**
 - Metriken, Coding Standards, ...
 - **formale Methoden**
 - Model Checking, ...
 - **dynamisches Testen**
 - Black-Box, White-Box, Regressionstests, ...



Warum Testen?

- verschiedene Möglichkeiten, Aussagen über Programme zu treffen:
 - **informelle Methoden**
 - Inspection, Review, Walkthrough, ...
 - **analytische Methoden**
 - Metriken, Coding Standards, ...
 - **formale Methoden**
 - Model Checking, ...
 - **dynamisches Testen**
 - Black-Box, White-Box, Regressionstests, ...
 - Verhalten eines Programms beurteilen
 - ➔ Programm ausführen
 - formale Methoden sind oft sehr mühsam, aufwendig, unmöglich, ...
-
- The diagram consists of four arrows pointing from the method categories to two text boxes. The top arrow points from 'informelle Methoden' to the box 'Aussagen über die Qualität'. The second arrow from the left points from 'analytische Methoden' to the same box. The third arrow from the left points from 'formale Methoden' to the box 'Aussagen über das Verhalten'. The fourth arrow from the left points from 'dynamisches Testen' to the same box 'Aussagen über das Verhalten'.

Testarten

- Testfälle in den Phasen der SW-Entwicklung
- Black-Box vs. White-Box



Tests nach den Phasen der SW-Entwicklung

- **Modultest** (engl. *module testing*)
Diskrepanzen zwischen der Implementierung und der im Entwurf / in der Spezifikation festgelegten Funktion / Schnittstelle
- **Integrationstest** (engl. *integration testing*)
Probleme beim Zusammenspiel mehrerer Module
- **Systemtest** (engl. *system testing*)
Black-Box-Test: tatsächliche Leistung vs. geforderte Leistung hinsichtlich Vollständigkeit, Volumen, Stresstest und Leistung
- **Abnahmetest** (engl. *acceptance testing*)
Erfüllt das Produkt den Anforderungen des Auftraggebers hinsichtlich Korrektheit, Robustheit, Performanz und Dokumentation



Black-Box vs. White-Box

■ Black-Box Testing

- keine Kenntnis der internen Struktur
- Testfälle basieren auf Spezifikation, Programmcode wird ignoriert
- synonym: *functional, data-driven, i/o-driven*

→ **Frage:** Wurden alle Anforderungen implementiert?

■ White-Box Testing

- Kenntnis der internen Struktur zwingend erforderlich
- Testfälle basieren auf Programmcode, Spezifikation wird ignoriert
- synonym: *structured, glass-box, logic-driven, path-oriented*

→ **Frage:** Wurden nur Anforderungen implementiert?



Problem: Black-Box Testing

- Beispiel OSEK OS:
 - 4 Conformance Klassen: BCC1, BCC2, ECC1, ECC2
 - 3 Scheduling Verfahren: NON, MIXED, FULL
 - 2 Statusklassen: STANDARD, EXTENDED
 - 24 Varianten für jeden Testfall
- kein Wissen über die interne Struktur vorhanden
 - Parameter könnten sich gegenseitige beeinflussen
 - alle Kombinationen müssen getestet werden:
kombinatorische Explosion
- Kombination mit White-Box Testing
 - Unabhängigkeit der Parameter kann evtl. sicher gestellt werden
 - Reduktion der Testfälle bzw. deren Varianten

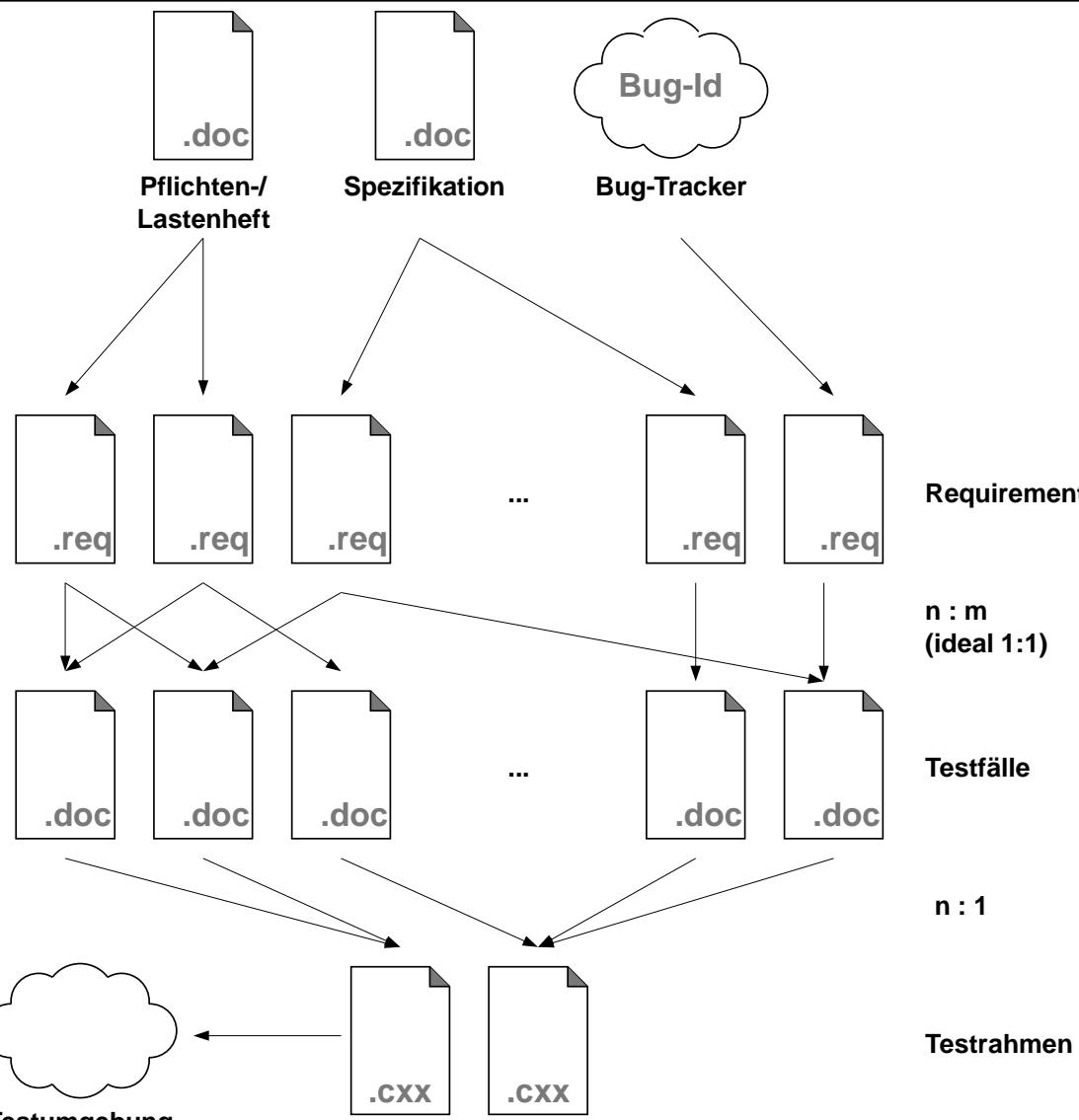


Wo kommen Testfälle her?

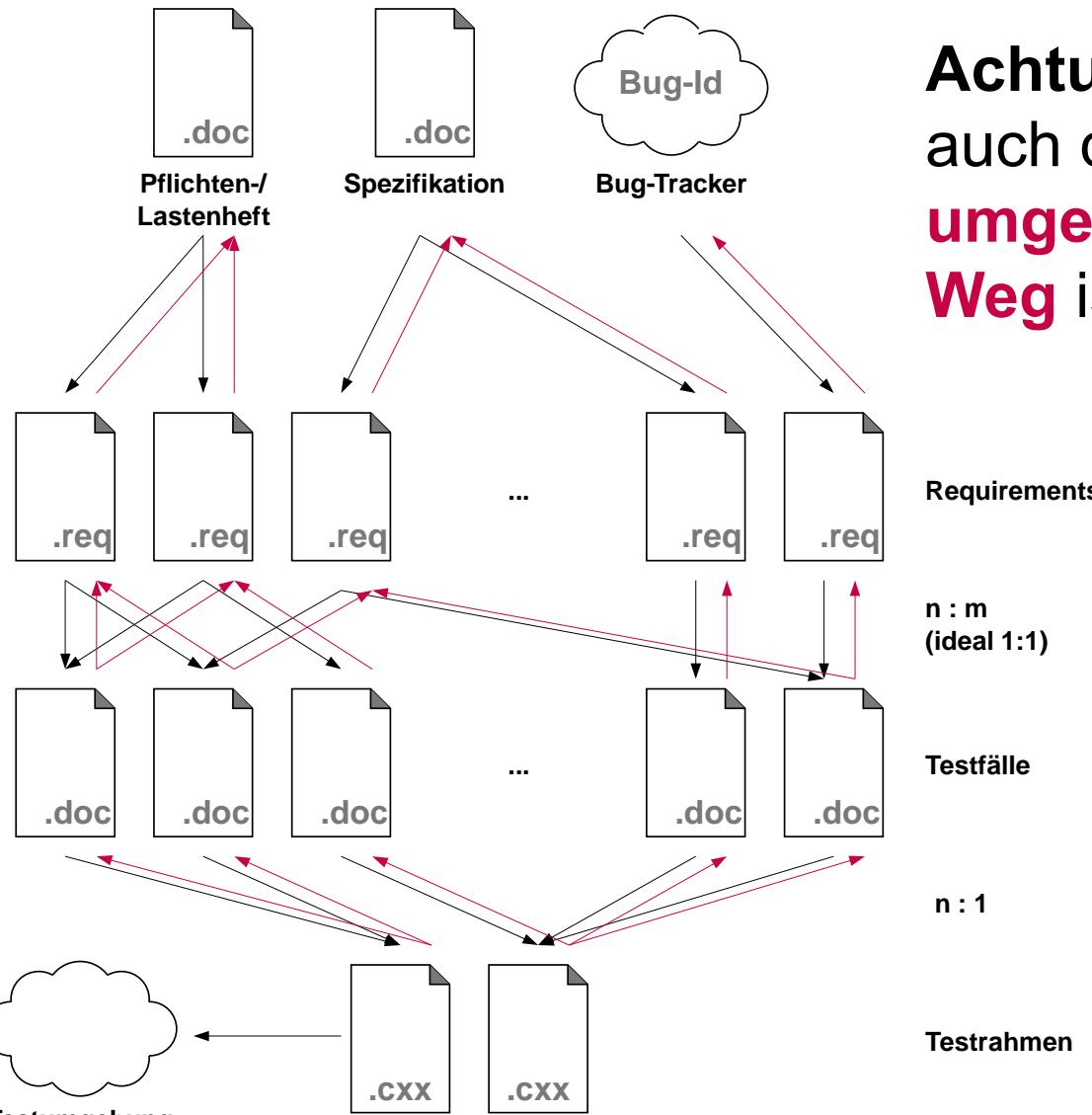
- Prozess
- Konstruktionshilfen für
 - Black-Box Testing
 - White-Box Testing



Prozess



Prozess



Achtung:
auch der
umgekehrte
Weg ist wichtig!!!

Konstruktionshilfen: Black-Box

- **Äquivalenzklassentest** (engl. *partition testing*)

Eingabewerte, die zu identischen Ergebnissen führen sollen, werden zu Äquivalenzklassen zusammengefasst, z.B. Klassifikationsbäume (Daimler-Chrysler). Ableitung der Äquivalenzklassen aus der **Spezifikation**.

- **Grenzwerttest** (engl. *boundary value testing*)

Eingabewerte an den Grenzen der Äquivalenzklassen

- **Cause-Effect-Graphing**

Eingabedaten ausgehend von Ursache-Wirkung-Überlegungen

- **Error-Guessing**

Aus Erfahrung erwarteter Fehler, bezogen auf die Spezifikation

- **Zufallstest** (engl. *random testing*)

Zufällige Eingabewerte, die einer bestimmten Verteilung gehorchen, Simulation realer Eingabewerte



Konstruktionshilfen: White-Box

- **Äquivalenzklassentest** (engl. *partition testing*)
Eingabewerte, die zu identischen Ergebnissen führen sollen, werden zu Äquivalenzklassen zusammengefasst, z.B. Klassifikationsbäume (Daimler-Chrysler). Ableitung der Äquivalenzklassen anhand des **Programmcodes**.
- **Grenzwerttest** (engl. *boundary value testing*)
Eingabewerte an den Grenzen der Äquivalenzklassen
- **Mehrfachbedingungstest** (engl. *multiple condition testing*)
Bei Verzweigungen, die mehrere Bedingungen enthalten, werden alle Bedingungen getestet.



Wie gut hat man getestet?

- **minimale Anzahl** von Testfällen
- Testabdeckung – **Coverage**
 - funktionale Coverage
 - Code Coverage
 - Datenfluss Coverage



Minimale Anzahl von Testfällen

- **McCabe's Cyclomatic Complexity**
 - Maß für die Anzahl der unabhängigen Pfade durch ein Programm
 - untere Schranke für die Anzahl der Testfälle

- **Function-Point-Metrik**
 - 1) Zähle Funktionen und zu verarbeitende Daten
→ unjustierter Function-Point-Wert
 - 2) Bewertung bestimmter, nicht-funktionaler Systemeigenschaften
→ justierter Function-Point-Wert
 - 3) Bezug des justierten Function-Point-Wertes mit Referenzdaten
→ Aufwandsabschätzung
 - McCabe: justierter Function-Point-Wert * 1,2
→ untere Schranke für die Anzahl der Testfälle



Funktionale Coverage

- wurden alle Anforderungen getestet
 - existiert zu jeder Anforderung mindestens ein Testfall
- Requirement Tracing!



Code Coverage (1)

- welcher Anteil des Programmcodes wurde getestet?
- keine Testfälle an sich, Maß für die Testabdeckung
- **Statement Coverage** $Sc = s / S$
 - s = Anzahl der erreichten Statements
 - S = Anzahl aller Statements
 - findet
 - nicht erreichbaren Code
 - nicht getesteten Code
 - vom Compiler nicht geprüften Code
- **Branch Coverage** $Bc = b / B$
 - b = Anzahl ausgewerteter Verzweigungsmöglichkeiten
 - B = Anzahl aller Verzweigungsmöglichkeiten
 - Structured Programming:
100% Branch Coverage → 100% Statement Coverage



Code Coverage (2)

■ Path Coverage $P_c = p / P$

- p = Anzahl getesterter Pfade durch ein Programm
- P = Anzahl aller Pfade durch ein Programm
- 100% Path Coverage impliziert 100% Branch Coverage
- kombinatorische Explosion: sehr aufwendig
- Beschränkung auf nicht-pathologische Pfade

■ Conditional Coverage $C_c = c / C$

- c = Anzahl ausgewerteter logischer Entscheidungen
- C = Anzahl aller logischen Entscheidungen
- Ähnlich, aber nicht gleich Path Coverage → Beispiel:

```
if(A() || B()) {  
    ...  
} else {  
    ...  
}
```

A()	B()
false	false
true	false
false	true
true	true

} 100% Path Coverage
50% Conditional Coverage



Datenfluss Coverage (1)

- jede Variable ist definiert durch
 - Definition (**Definition**)
 - Verwendung (**Use**)
- **DU-Path**

Pfad in der Programmausführung von der Definition einer Variablen bis zu ihrer Verwendung ohne erneute Definition derselben Variable



Datenfluss Coverage (2)

- **all-defs**
mindestens ein Pfad von jeder Definition zu mindestens einer Verwendung
- **all-p-uses / some-c-uses > all-defs**
mindestens ein Pfad von jeder Definition zu jeder erreichbaren Verwendung innerhalb von Bedingungen oder mindestens einer Berechnung, falls keine erreichbare Verwendung innerhalb einer Bedingung existiert
- **some-p-uses / all-c-uses > all-defs**
analog zu all-p-uses / some-c-uses
- **all-uses > all-p-uses / some-c-uses | some-p-uses / all-c-uses**
mindestens ein Pfad von jeder Definition zu jeder erreichbaren Verwendung
- **all-DU-paths**
alle DU-Pfade für jede Definition



Spezifikation von Testfällen

- die Spezifikation enthält
 - **Testfallbezeichner**
 - **Requirements**, die getestet werden
 - **Vorbedingungen, Eingabedaten**
 - **erwartetes Ergebnis**



Spezifikation von Testfällen

■ Beispiel ProOSEK Testfall:

```
TESTCASE SetRelAlarm2
@SCHEDULE n,m,t
@CC BCC1, BCC2, ECC1, ECC2
@STATUS s,e
@SCENARIO
```

Bezeichner

Aufruf von SetRelAlarm() mit einem bereits aktivierten Alarm, der bei seinem Ablauen einen Task aktiviert.

@RESULT

Der Aufruf liefert E_OS_STATE.

@REQUIREMENTS

```
//REQ: Kernel.API.Alarms.DeclareAlarm, TEST
//REQ: Kernel.API.Alarms.SetRelAlarm.API, TEST
//REQ: Kernel.API.Alarms.SetRelAlarm.Task, TEST
//REQ: Kernel.API.Alarms.SetRelAlarm.ISRC2, TEST
//REQ: Kernel.API.Alarms.SetRelAlarm.AlreadyInUse, TEST
TESTCASE END
```



Spezifikation von Testfällen

■ Beispiel ProOSEK Testfall:

Vorbedingungen /
Eingabewerte

```
TESTCASE SetRelAlarm2
@SCHEDULE n,m,t
@CC BCC1, BCC2, ECC1, ECC2
@STATUS s,e
@SCENARIO
Aufruf von SetRelAlarm() mit einem bereits aktivierten
Alarm, der bei seinem Ablauen einen Task aktiviert.
@RESULT
Der Aufruf liefert E_OS_STATE.
@REQUIREMENTS
//REQ: Kernel.API.Alarms.DeclareAlarm, TEST
//REQ: Kernel.API.Alarms.SetRelAlarm.API, TEST
//REQ: Kernel.API.Alarms.SetRelAlarm.Task, TEST
//REQ: Kernel.API.Alarms.SetRelAlarm.ISRC2, TEST
//REQ: Kernel.API.Alarms.SetRelAlarm.AlreadyInUse, TEST
TESTCASE END
```



Spezifikation von Testfällen

■ Beispiel ProOSEK Testfall:

```
TESTCASE SetRelAlarm2
@SCHEDULE n,m,f
@CC BCC1, BCC2, ECC1, ECC2
@STATUS s,e
@SCENARIO
Aufruf von SetRelAlarm() mit einem bereits aktivierten
Alarm, der bei seinem Ablauen einen Task aktiviert.
@RESULT
Der Aufruf liefert E_OS_STATE.
@REQUIREMENTS
//REQ: Kernel.API.Alarms.DeclareAlarm, TEST
//REQ: Kernel.API.Alarms.SetRelAlarm.API, TEST
//REQ: Kernel.API.Alarms.SetRelAlarm.Task, TEST
//REQ: Kernel.API.Alarms.SetRelAlarm.ISRC2, TEST
//REQ: Kernel.API.Alarms.SetRelAlarm.AlreadyInUse, TEST
TESTCASE END
```

erwartetes
Ergebnis



Spezifikation von Testfällen

■ Beispiel ProOSEK Testfall:

```
TESTCASE SetRelAlarm2
@SCHEDULE n,m,f
@CC BCC1, BCC2, ECC1, ECC2
@STATUS s,e
@SCENARIO
Aufruf von SetRelAlarm() mit einem bereits aktivierten
Alarm, der bei seinem Ablauen einen Task aktiviert.
```

```
@RESULT
Der Aufruf liefert E_OS_STATE.
@REQUIREMENTS
```

```
//REQ: Kernel.API.Alarms.DeclareAlarm, TEST
//REQ: Kernel.API.Alarms.SetRelAlarm.API, TEST
//REQ: Kernel.API.Alarms.SetRelAlarm.Task, TEST
//REQ: Kernel.API.Alarms.SetRelAlarm.ISRC2, TEST
//REQ: Kernel.API.Alarms.SetRelAlarm.AlreadyInUse, TEST
TESTCASE END
```

geprüfte
Requirements



Implementierung von Testfällen

- Testrahmen
- Testumgebung



Testrahmen

- genau definierte Anwendung
- enthält Implementierung eines oder mehrerer Testfälle
- Ablauf wird durch die Testumgebung gesteuert



Testrahmen

■ Beispiel ProOSEK Testfall:

```
...
TASK (task1) {
    ...
    /* @TESTCASE GetAlarmBase2 */
    if((status = GetAlarmBase(alarm1,&base)) != E_OK) {
        PanicStatus("GetAlarmBase() returned wrong value!",status);
    }
    if(base.maxallowedvalue != 255 || base.ticksperbase != 50) {
        Panic("GetAlarmBase() returned wrong alarmbase for Alarm1!");
    }

    sequence[counter] = 'a';
    counter++;

    /* @TESTCASE SetAbsAlarm9 */
    if((status = SetAbsAlarm(alarm1,255,0)) != E_OK) {
        PanicStatus("SetAbsAlarm() returned wrong value!",status);
    }
    ...
    TerminateTask();
}
...
```



Testrahmen

■ Beispiel ProOSEK Testfall:

```
...
TASK (task1) {
    ...
    /* @TESTCASE GetAlarmBaseZ */
    if((status = GetAlarmBase(alarm1,&base)) != E_OK) {
        PanicStatus("GetAlarmBase() returned wrong value!",status);
    }
    if(base.maxallowedvalue != 255 || base.ticksperbase != 50) {
        Panic("GetAlarmBase() returned wrong alarmbase for
Alarm1!");
    }

    sequence[counter] = 'a';
    counter++;

    /* @TESTCASE SetAbsAlarm9 */
    if((status = SetAbsAlarm(alarm1,255,0)) != E_OK) {
        PanicStatus("SetAbsAlarm() returned wrong value!",status);
    }
    ...
    TerminateTask();
}
...
```

Testfallbezeichner



Testrahmen

■ Beispiel ProOSEK Testfall:

Testfallimplementierung

```
...
TASK (task1) {
    ...
    /* @TESTCASE GetAlarmBase2 */
    if((status = GetAlarmBase(alarm1,&base)) != E_OK) {
        PanicStatus("GetAlarmBase() returned wrong value!",status);
    }
    if(base.maxallowedvalue != 255 || base.ticksperbase != 50) {
        Panic("GetAlarmBase() returned wrong alarmbase for
Alarm1!");
    }

    sequence[counter] = 'a';
    counter++;

    /* @TESTCASE SetAbsAlarm9 */
    if((status = SetAbsAlarm(alarm1,255,0)) != E_OK) {
        PanicStatus("SetAbsAlarm() returned wrong value!",status);
    }
    ...
    TerminateTask();
}

...
```



Testrahmen

■ Beispiel ProOSEK Testfall:

```
...
TASK (task1) {
    ...
    /* @TESTCASE GetAlarmBase2 */
    if((status = GetAlarmBase(alarm1,&base)) != E_OK) {
        PanicStatus("GetAlarmBase() returned wrong value!",status);
    }
    if(base.maxallowedvalue != 255 || base.ticksperbase != 50) {
        Panic("GetAlarmBase() returned wrong alarmbase for
Alarm1!");
    }

    sequence[counter] = 'a',
    counter++;

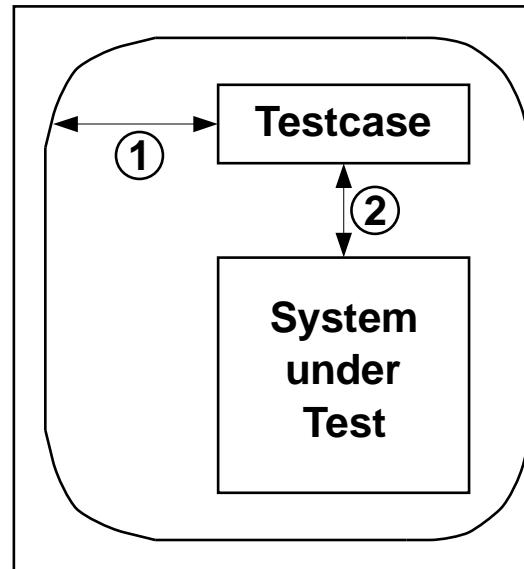
    /* @TESTCASE SetAbsAlarm9 */
    if((status = SetAbsAlarm(alarm1,255,0)) != E_OK) {
        PanicStatus("SetAbsAlarm() returned wrong value!",status);
    }
    ...
    TerminateTask();
}
...
```

Ablaufsteuerung



Testumgebung

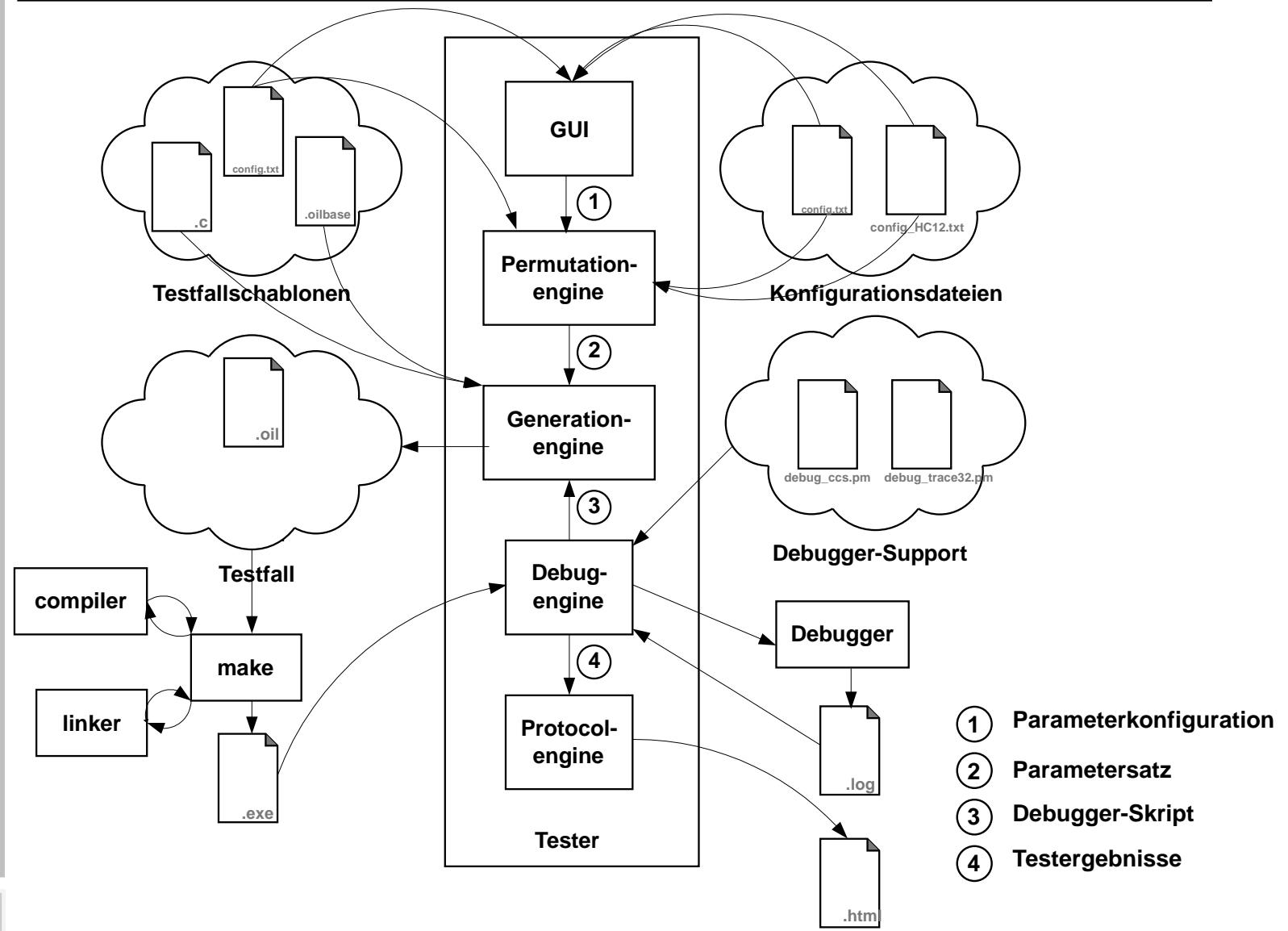
- Bereitstellung einer Test-API
 - Kontrolle des Testablaufs
- **Ausführung** der Testfälle
- **Auswertung** und **Protokollierung** der Ergebnisse



① **Test-API**

② **SUT-API**

Testumgebung – ProOSEK Testsuite



Testumgebung – ProOSEK Testsuite

- Umfang:
 - > 40 Perl-Module
 - > 15 allgemeine, architektur- und testfallspezifische Konfigurationsdateien
 - > 20 architektspezifische Header- und Implementierungsdateien
 - sehr viele Testfälle, allgemein und architektspezifisch
 - > 20000 LOC Perl + ASCII
- Vorteil
 - hohes Maß an Generizität, leicht erweiterbar
- Problem
 - hohes Maß an Komplexität
 - wer testet die TestSuite?



Performanztests

- Speicherbedarf
- Laufzeit



Speicherbedarf

- Wie viel Speicher wird benötigt?
 - Programmcode
 - Stack
 - Daten (lesbar / schreibbar)
- statische Auswertung des übersetzten Programms bzw. der Map-Datei
 - Speicherbedarf des Programmcodes
 - Speicherbedarf des Stacks (Worst Case, Average Case, ...)
 - Speicherbedarf der Daten



Laufzeit (1)

■ statisch: *instruction counting*

- Laufzeiten der Instruktionen sind bekannt
→ Gesamtlaufzeit kann berechnet werden
- enorm schwierig, viele Faktoren müssen beachtet werden, um brauchbare Ergebnisse zu erzielen
 - Pipeline des Prozessors
 - Speicherhierarchie
 - Out-of-Order-Execution
 - Branch Prediction
 - Eingabedaten
 - ...



Laufzeit (2)

■ **dynamisch: Messung**

- Primitive: `start()`, `stop()`, `get_time()`
- Messungen immer mehrmals durchführen
 - Ergebnisse mitteln
 - Median
 - Standardabweichung, Varianz

■ **Achtung**

- Kontextwechsel
- geschachtelte Messungen
- Unterbrechungen
- Kalibrierung
- misst man auch wirklich die WCET?



Zusammenfassung

- **Warum testet man**
 - um das Verhalten von Software zu erproben
- **Welche Testarten gibt es?**
 - Modul-, Integrations-, System-, Abnahmetests
 - Black-Box, White-Box Tests
- **Wo kommen Testfälle her?**
 - Requirement Engineering
 - Konstruktionshilfen
- **Hat man ausreichend getestet?**
 - minimale Anzahl von Testfällen
 - Coverage
- **Testfallspezifikation**
- **Testfallimplementierung**
 - Testrahmen, Testumgebung



Ergebnis

- Einfach Testumgebung: make <testcase>
 - führt Testfall aus
 - protokolliert Ergebnisse
- funktionale Tests
 - funktionaler Test von mindestens 50% der Module
 - Spezifikation & Implementierung
 - keine Coverage-Messungen
- Performanztests
 - keine Messung des Speicherbedarfs
 - Messung aller relevanten WCETs mit AbsInt aiT
 - Vernachlässigung des Betriebssystems

