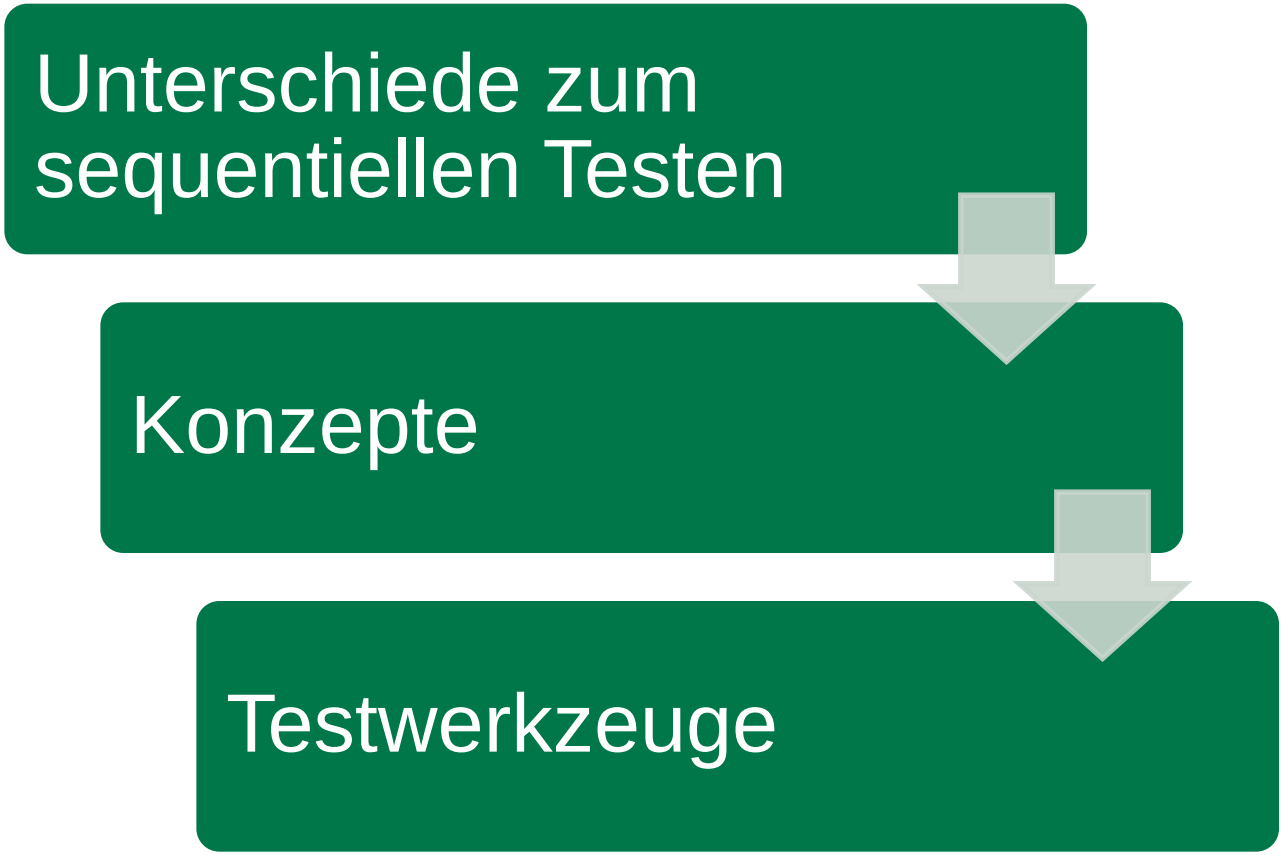
Multicore überall
Testwerkzeuge für nebenläufige Anwendungen

Oliver Denninger

Entwicklertag 2013, Karlsruhe, 6. Juni

Testen nebenläufiger Anwendungen



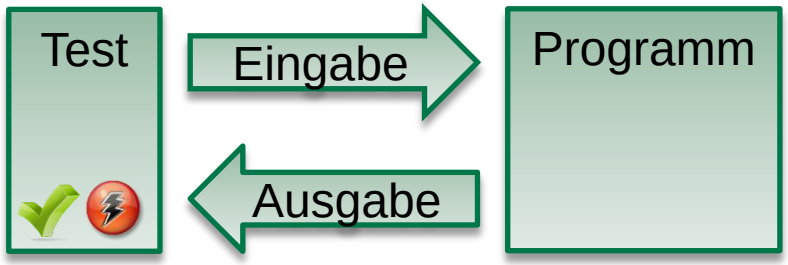
Nebenläufige Anwendungen

- Programmablauf wird stark vom zeitlichen Verhalten beeinflusst
  - Software, z.B. Scheduling
  - Hardware, z.B. Cache-Misses

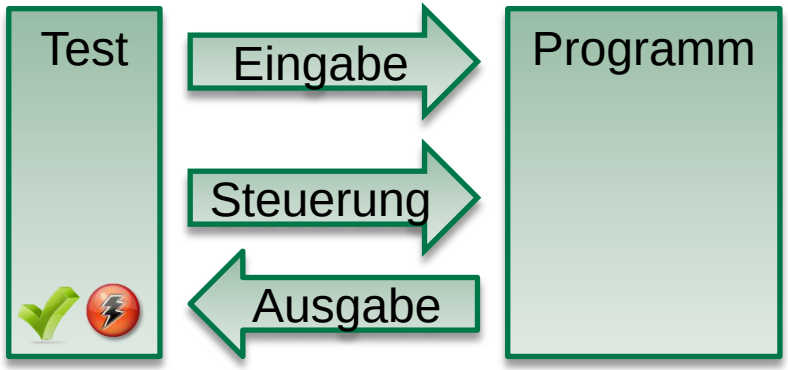
- zeitliches Verhalten legt fest …
  - in welcher Reihenfolge Anweisungen ausgeführt werden
  - welche Anweisungen ausgeführt werden („busy waiting“)

Testen

- Kombinationen von Eingaben und Ausgaben prüfen



- Ausgaben nebenläufiger Anwendungen hängen von den Eingaben **und vom zeitlichen Verhalten** ab
- Test muss zeitliches Verhalten einbeziehen



Thread Interleavings

- Anzahl möglicher Interleavings steigt exponentiell mit Programmgröße
- nur wenige Interleavings führen zu Fehlern

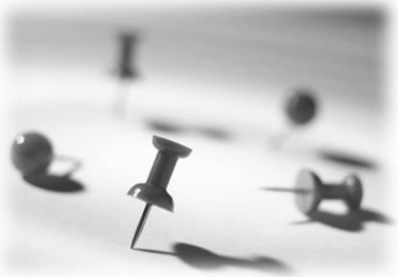
- in der Praxis treten oft ähnliche Interleavings auf, da z.B. Scheduler mit festen Regeln arbeiten

➢ für vorhandene Testverfahren ist die Wahrscheinlichkeit einen Nebenläufigkeitsfehler zu entdecken sehr gering

Thread Interleavings beeinflussen

- in der Praxis durch Beeinflussung des Schedulings
- möglichst viele verschiedene Interleavings ausführen um unbekannte **Fehler zu finden**

- gezielt bestimmte Interleavings ausführen um **Fehler zu reproduzieren**



- alle möglichen Interleavings ausführen?

Überblick der Konzepte

- einfache, wiederholte Ausführung
  - wiederholte Ausführung von Anwendungen bzw. Tests
  - Stress-Test
- Ausführung unterschiedlicher Interleavings
  - *ConTest von IBM (Java bzw. C mit PThreads)*
- gezielte Ausführung bestimmter Interleavings
  - *MultithreadedTC (Java)*
  - *IMUnit (Java)*
- Testen aller relevanter Interleavings
  - *CHESS von Microsoft (.NET / Win32)*
  - *cJUnit (Java)*




1. Wiederholte Ausführung

- Anwendungen bzw. Tests mit sehr vielen Wiederholungen ausführen
  - einige Test-Frameworks bieten Unterstützung (TestNG, jMock, diverse JUnit-Erweiterungen)
  - weder effizient, noch zuverlässig

- Stress-Test
  - Anwendung unter hoher Last ausführen
  - bei langlaufenden Anwendungen ergeben sich viele Wiederholungen von Programmabschnitten

2. Unterschiedliche Interleavings ausführen

- Scheduler während der Ausführung beeinflussen
- unterschiedliche Interleavings erzwingen
- oft basierend auf Heuristiken

ConTest

- bei wiederholter Ausführung einer Anwendung werden unterschiedliche Interleavings erzwungen
  - Instrumentierung des Java-Bytecodes
  - fügt zufällige **`sleep()`** oder **`yield()`** Aufrufe bei Synchronisierungsoperationen ein
- nutzbar für Anwendungen oder multi-threaded Tests
- lediglich ConTest.jar in Classpath aufnehmen
- erhöht Wahrscheinlichkeit Nebenläufigkeitsfehler zu finden

3. Gezielt bestimmte Interleavings ausführen

- Entwickler gibt aus seiner Sicht „problematische“ Interleavings an
- fehlerauslösende Interleavings reproduzieren
- Einfügen von Sleep()
  - sehr einfacher Ansatz
  - Tests dauern länger
  - beeinflusst Scheduling, allerdings nicht exakt vorhersehbar

Beispiel mit “sleep”

```
WorkQueue wq = new WorkQueue(1); Work work = new Work();

public void testPutTake() throws InterruptedException {
    Thread master = new Thread() {
        public void run() {
            try {
                Thread.sleep(100);
                wq.put(work);
            } catch (InterruptedException ie) {}
        }
    };
    master.start();
    Work w = wq.take();
    master.join();
    assertThat(“wq empty”, wq.isEmpty());
    assertThat(w, equalTo(work));
}
```

Beispiel mit “sleep”

```
WorkQueue wq = new WorkQueue(1); Work work = new Work();

public void testPutTake() throws InterruptedException {
    Thread master = new Thread() {
        public void run() {
            try {
                Thread.sleep(100);
                wq.put(work);
            } catch (InterruptedException ie) {}
        }
    };
    master.start();
    Work w = wq.take();
    master.join();
    assertThat(“wq empty”, wq.isEmpty());
    assertThat(w, equalTo(work));
}
```

MultithreadedTC

- MultithreadedTC garantiert, dass ein Test im vorgegebenen Interleaving ausgeführt wird
- zeitgleiches Blockieren aller Threads löst „Ticks“ aus
- Threads können auf bestimmte „Ticks“ warten
- entwickelt um „kleine Datenstrukturen“ zu testen
  - Idee, dass die problematischen Interleavings von Hand spezifiziert werden können
- kann als JUnit-Erweiterung genutzt werden

MultithreadedTC – Beispiel

```
WorkQueue wq; Work work, retrievedWork;

public void initialize() {
    wq = new WorkQueue(1);
    work = new Work();
}
public void thread1() throws InterruptedException {
    waitForTick(1);
    wq.put(work);
}
public void thread2() throws InterruptedException {
    retrievedWork = wq.take();
    assertTick(1);
}
public void finished() {
    assertThat(wq.isEmpty()); assertThat(retrievedWork, equalTo(work));
}
```

MultithreadedTC – Beispiel

```
WorkQueue wq; Work work, retrievedWork;

public void initialize() {
    wq = new WorkQueue(1);
    work = new Work();
}
public void thread1() throws InterruptedException {
    waitForTick(1);
    wq.put(work);
}
public void thread2() throws InterruptedException {
    retrievedWork = wq.take();
    assertTick(1);
}
public void finished() {
    assertThat(wq.isEmpty()); assertThat(retrievedWork, equalTo(work));
}
```

MultithreadedTC – Beispiel

```
WorkQueue wq; Work work, retrievedWork;

public void initialize() {
    wq = new WorkQueue(1);
    work = new Work();
}
public void thread1() throws InterruptedException {
    waitForTick(1);
    wq.put(work);
}
public void thread2() throws InterruptedException {
    retrievedWork = wq.take();
    assertTick(1);
}
public void finished() {
    assertThat(wq.isEmpty()); assertThat(retrievedWork, equalTo(work));
}
```

MultithreadedTC – Beispiel

```
WorkQueue wq; Work work, retrievedWork;

public void initialize() {
    wq = new WorkQueue(1);
    work = new Work();
}
public void thread1() throws InterruptedException {
    waitForTick(1);
    wq.put(work);
}
public void thread2() throws InterruptedException {
    retrievedWork = wq.take();
    assertTick(1);
}
public void finished() {
    assertThat(wq.isEmpty()); assertThat(retrievedWork, equalTo(work));
}
```

IMUnit

- versucht Nachteile von MultithreadedTC zu beseitigen
- statt globalen „Ticks“ kommen frei wählbare Ereignisse zum Einsatz
- der Entwickler spezifiziert wann Ereignisse ausgelöst werden und welche Threads darauf warten
- JUnit-Erweiterung

IMUnit – Beispiel

```
WorkQueue wq = new WorkQueue(1); Work work = new Work();
@Test
@Schedule("[beforeTake]->beforePut")
public void testPutTake() throws InterruptedException {
    Thread master = new Thread() {
        public void run() {
            try {
                fireEvent("beforePut");
                wq.put(work);
                fireEvent("afterPut");
            } catch (InterruptedException ie) {}
    }};
    master.start();
    fireEvent("beforeTake");
    Work w = wq.take();
    fireEvent("afterTake");
    master.join();
    assertThat("wq empty", wq.isEmpty()); assertThat(w, equalTo(work));
}
```

IMUnit – Beispiel

```
WorkQueue wq = new WorkQueue(1); Work work = new Work();
@Test
@Schedule("[beforeTake]->beforePut")
public void testPutTake() throws InterruptedException {
    Thread master = new Thread() {
        public void run() {
            try {
                fireEvent("beforePut");
                wq.put(work);
                fireEvent("afterPut");
            } catch (InterruptedException ie) {}
    }};
    master.start();
    fireEvent("beforeTake");
    Work w = wq.take();
    fireEvent("afterTake");
    master.join();
    assertThat("wq empty", wq.isEmpty()); assertThat(w, equalTo(work));
}
```

IMUnit – Beispiel

```
WorkQueue wq = new WorkQueue(1); Work work = new Work();
@Test
@Schedule("[beforeTake]->beforePut")
public void testPutTake() throws InterruptedException {
    Thread master = new Thread() {
        public void run() {
            try {
                fireEvent("beforePut");
                wq.put(work);
                fireEvent("afterPut");
            } catch (InterruptedException ie) {}
    }};
    master.start();
    fireEvent("beforeTake");
    Work w = wq.take();
    fireEvent("afterTake");
    master.join();
    assertThat("wq empty", wq.isEmpty()); assertThat(w, equalTo(work));
}
```

4. Testen aller relevanter Interleavings

- möglichst alle Interleavings testen
- außer für triviale Anwendungen nicht möglich
- nicht alle möglichen Interleavings sind relevant, da an vielen Codestellen mögliche Kontextwechsel uninteressant sind
  - z.B. Zugriffe auf lokale Variablen
- relevante Teilmenge aller Interleavings ermitteln und vollständig überprüfen

CHESS

- wählt einige mögliche Stellen für Kontextwechsel und überprüft darauf basierend alle möglichen Interleavings
- durch wiederholte Ausführung können insgesamt mehr Kontextwechsel abgedeckt werden
- nutzbar für Anwendungen oder multi-threaded Tests
- Testen von Anwendungen bzw. großen Tests nur wenn wenige Stellen für Kontextwechsel berücksichtigt werden
- einsetzbar für .NET bzw. Win32-Anwendungen
- Integration in Visual Studio möglich

cJUnit

- berücksichtigt alle möglichen Kontextwechsel die für Java relevant sind
  - Synchronisierung und Zugriffe auf gemeinsame Variablen
- überprüft alle Interleavings
- für kleine, fokussierte Tests
- JUnit-Erweiterung

cJUnit – Beispiel

```
WorkQueue wq = new WorkQueue(1); Work work = new Work();

@ConcurrentTest(threadGroup=42)
public void testPut() throws InterruptedException {
    wq.put();
}

@ConcurrentTest(threadGroup=42)
public void testTake() throws InterruptedException {
    Work w = wq.take();
    assertThat("wq empty", wq.isEmpty());
    assertThat(w, equalTo(work));
}
```

cJUnit – Beispiel

```
WorkQueue wq = new WorkQueue(1); Work work = new Work();

@ConcurrentTest(threadGroup=42)
public void testPut() throws InterruptedException {
    wq.put();
}

@ConcurrentTest(threadGroup=42)
public void testTake() throws InterruptedException {
    Work w = wq.take();
    assertThat("wq empty", wq.isEmpty());
    assertThat(w, equalTo(work));
}
```

Verfügbarkeit der Testwerkzeuge (1)

- ConTest (IBM, Java)
  - verfügbar als kommerzieller “On Demand Innovation Service”
  - https://www.research.ibm.com/haifa/projects/verification/contest/

Verfügbarkeit der Testwerkzeuge (2)

- MultithreadedTC (Java)
  - http://code.google.com/p/multithreadedtc/
  - http://code.google.com/p/multithreadedtc-junit4/
- TickingTest (C#)
  - .NET Portierung von MultithreadedTC
  - http://code.google.com/p/donkirkby/wiki/TickingTest
- IMUnit (Java)
  - http://mir.cs.illinois.edu/imunit/
- ThreadWeaver (Java)
  - http://code.google.com/p/thread-weaver/
- CONCURRIT (C++)
  - http://code.google.com/p/concurrit/
- Awaitility (Java, asynchrone Operationen)
  - http://code.google.com/p/awaitility/

Verfügbarkeit der Testwerkzeuge

- CHESS (Microsoft, .NET, Win32)
  - aktuell nur nicht-kommerzielle Lizenz
  - http://chesstool.codeplex.com/
- cJUnit (Java)
  - open source
  - https://github.com/szeder/cJUnit/

denninger@fzi.de