Nebenläufigkeitsfehler in C# den Garaus machen!

Luc Bläser

Hochschule für Technik Rapperswil

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

HSR Concurrency Lab
Prof. Dr. Luc Bläser

Parallel 2017 Konferenz
31. März 2017, Heidelberg

Nebenläufigkeitsfehler

- Fehler, die es im Sequentiellen nicht gibt
- Nicht-deterministisches Auftreten
  - Je nach verzahnter oder paralleler Thread-Schedulierung
- Schwierig zu finden, reproduzieren und testen

Fehlerarten

- Race Conditions
- Data Races
- Deadlocks
- Livelocks
- Starvation

Race Condition

- Mehrere Threads greifen gemeinsame Ressourcen ohne genügende Synchronisation zu

- Mögliches Fehlverhalten oder falsche Resultate
  - Je nach Verzahnung oder Gleichzeitigkeit

Ursache ist oft ein Data Race, aber nicht immer!

Data Race

- Unsynchronisierte nebenläufige Speicherzugriffe
  - Selbe Variable oder Array Element
  - Mindestens ein schreibender Zugriff
- Formaler Fehler

```
int balance;
```

Thread 1

```
balance = 100;
```

Thread 2

```
while (balance == 0) {}
```

Endlosschlaufe möglich

Weitere Fälle

Thread 1

```
balance += 100;
```

Thread 2

```
balance += 50;
```

Lost Updates

Thread 1

```
if (balance >= amount) {
  balance -= amount;
}
```

Thread 2

```
if (balance >= amount) {
  balance -= amount;
}
```

Negative Saldi u.a.

Race Condition ohne Data Race

```
class BankAccount {
  private readonly object sync = new object();
  private int balance;

  public Balance {
     get { lock(sync) { return balance; } }
     set { lock(sync) { balance = value; } }
  }
}
```

Mehrere Threads

```
Balance += 100;
```

Nicht atomar

Deadlock

- Mehrere Threads sperren sich gegenseitig, so dass keiner fortschreiten kann
  - Geschachtelte Locks
  - Zyklische Warteabhängigkeit

Thread 1

```
lock(a) {
  lock (b) { … }
}
```

Thread 2

```
lock(b) {
  lock(a) { … }
}
```

Thread 1 sperrt a, 2 sperrt b => Deadlock

Livelock

- Spezialfall des Deadlocks
- Threads haben sich gegenseitig permanent blockiert
  - Laufen jedoch im Kreis während des Wartens
  - Verbrauchen CPU während der ewigen Blockade

Thread 1

```
a = false;
while (!b) { }
…
a = true;
```

Thread 2

```
b = false;
while (!a) { }
…
b = true;
```

Livelock möglich

Starvation

- Ein Thread wird beliebig lange aufgehalten
  - Obwohl er immer wieder die Chance hätte
  - Andere Threads können ihn dauernd überholen
- Nie hoffnungslos blockiert (kein Livelock)

```
do {
  success = account.withdraw(100);
} while (!success);
```

Starvation möglich

Fehler vermeiden

- Software Architektur
  - Nebenläufigkeitsdesign
- Unit & Integration Testing
  - Nebenläufigkeitstests
- Analyse-Tools
  - Dynamisch oder statisch

Nebenläufigkeitsdesign

- Klärungsbedarf
  - Welche Threads greifen auf welche Objekte zu?
  - Welche Objekte müssen Thread-sicher sein?
  - Sind die Sperren hierarchisch/linear geordnet?

\+ Fehler proaktiv vermeiden

– Design muss eingehalten werden

Nebenläufigkeitstest

- Nebenläufige Stress Tests
  - Viele Threads rufen Operationen auf, Endzustand prüfen
- Kontrollierte Verzahnungen
  - Bewusst Synchronisation in Tests einsetzen

\+ Wertvolle Findings
– Sporadisches Auftreten
– Nicht reproduzierbar
– Aufwendig

Concurrent Unit Test

```
[TestMethod]
[Timeout(TestTimeout)]
public void TestConcurrentDeposits() {
  const int N = 100;
  var account = new BankAccount();
  var threads = new List<Thread>();
  for (int count = 0; count < N; count++) {
    var thread = new Thread(() => account.Deposit(1));
    thread.Start();
  }
  foreach (var thread in threads) {
    thread.Join();
  }
  Assert.AreEqual(N, account.Balance);
}
```

Timeout im Falle von Deadlocks/Blockaden

Gestartete Threads immer joinen

Schlusszustand prüfen

Exceptions in Threads mit Global Exception Handler erkennen

Analyse Tool

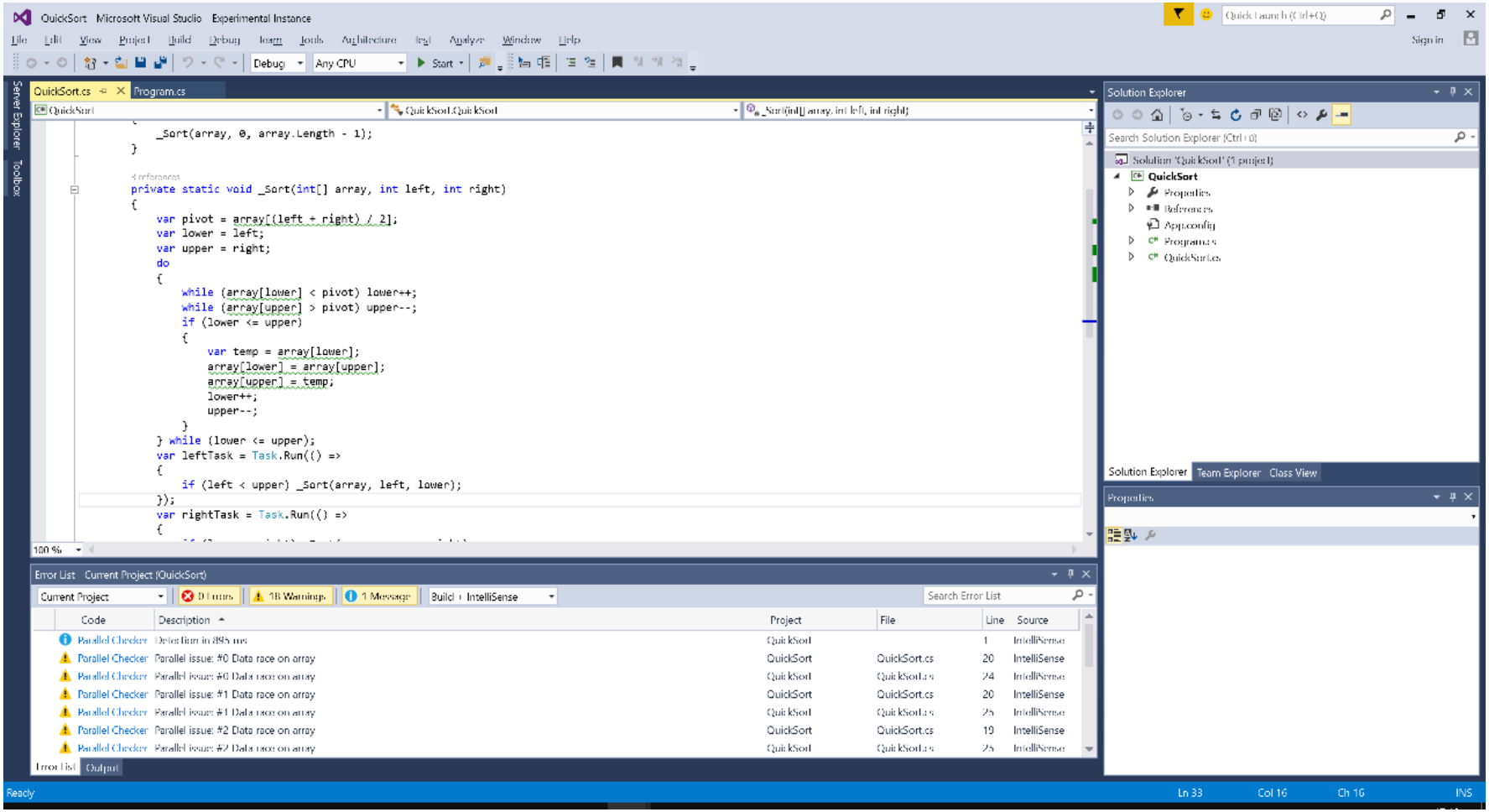
- Systematische Fehler-Erkennung
- Dynamisch
  - Beim laufenden Programm
  - Z.B. Intel Inspector, CHESS (ehemals)
- Statisch
  - Programmcode ohne Ausführung
  - Z.B. Pathfinder für Java, neues Tool

\+ Seltene Fälle erkennbar
– False Positives oder False Negatives
– Erkennen nur bestimmte Fehler (Data Races, Deadlocks)

HSR Parallel Checker

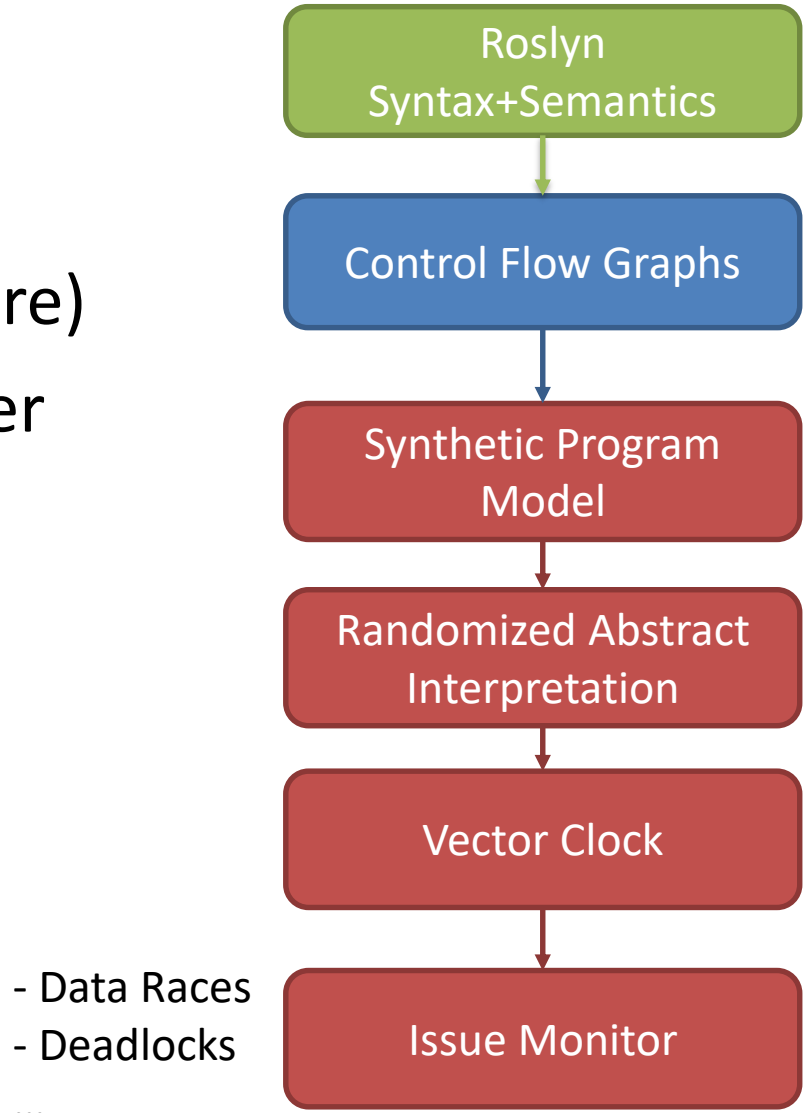
- Erkennung von Nebenläufigkeitsfehler in C#
  - Data Races
  - Deadlocks
- Markierung in Visual Studio während Coding
  - Statische Analyse
  - Basierend auf Roslyn
- Ziel: Schnelle und präzise Warnungen
  - Dafür evtl. nicht alle Fehler

Demo


QuickSort Microsoft Visual Studio Experimental Instance
File Edit View Project Build Debug Team Tools Architecture Test Analyze Window Help
Debug
Any CPU
Start
QuickSort.cs
Program.cs
_Sort(array, 0, array.Length - 1);
}
4 references
private static void _Sort(int[] array, int left, int right)
{
var pivot = array[(left + right) / 2];
var lower = left;
var upper = right;
do
{
while (array[lower] < pivot) lower++;
while (array[upper] > pivot) upper--;
if (lower <= upper)
{
var temp = array[lower];
array[lower] = array[upper];
array[upper] = temp;
lower++;
upper--;
}
} while (lower <= upper);
var leftTask = Task.Run(() =>
{
if (left < upper) _Sort(array, left, lower);
});
var rightTask = Task.Run(() =>
{
100 %
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'QuickSort' (1 project)
QuickSort
Properties
References
App.config
Program.cs
QuickSort.cs
Solution Explorer Team Explorer Class View
Properties
Error List Current Project (QuickSort)
Current Project
0 Errors
18 Warnings
1 Message
Build + IntelliSense
Search Error List
Code Description Project File Line Source
Parallel Checker Detection in 895 ms QuickSort 1 IntelliSense
Parallel Checker Parallel issue: #0 Data race on array QuickSort QuickSort.cs 20 IntelliSense
Parallel Checker Parallel issue: #0 Data race on array QuickSort QuickSort.cs 24 IntelliSense
Parallel Checker Parallel issue: #1 Data race on array QuickSort QuickSort.cs 20 IntelliSense
Parallel Checker Parallel issue: #1 Data race on array QuickSort QuickSort.cs 25 IntelliSense
Parallel Checker Parallel issue: #2 Data race on array QuickSort QuickSort.cs 19 IntelliSense
Parallel Checker Parallel issue: #2 Data race on array QuickSort QuickSort.cs 25 IntelliSense
Error List Output
Ready
Ln 33 Col 16 Ch 16 INS

Analyseverfahren

- Abstract Interpretation
- Randomisierte Schedules
- Vector Clock (Happens-Before)
- Bounds auf Zeit und Speicher



- Data Races
- Deadlocks
- …

Eigenschaften

- Statische Analyse: Programm läuft nicht
- Präzise Fehler: Möglichst exakter Zustand
- Konservativ: Bei unbekanntem I/O etc.
- Schnelle Feedbacks: Paar Sekunden
- Reproduzierbar: Findet jeweils gleiche Fehler

- Aber unvollständig: Kann Fehler übersehen!

Sprachumfang

- Explizite Threads
- Monitor, diverse Synchronizationsprimitiven
- TPL Tasks
- Parallel.Invoke/For/ForEach
- Async/Await (mit und ohne UI)
- Volatile und Interlocked
- DLL, WPF, EXE, Test Assemblies
- Finalizers
- C# 6 & 7: Arrays, OO, Lambdas, Exceptions etc.

- Aktuelle Limitationen: LINQ & PLINQ, Unsafe

Verfügbarkeit

- Visual Studio Gallery
  - «HSR Parallel Checker» installieren
  - «Full Solution Analysis» in VS aktivieren
- Für VS 2017 und VS 2015
- Projekt-Webseite
  - http://parallel-checker.com
- Kostenfeie unbeschränkte Nutzung in VS
- Aktuell: Beta Version
  - Feedbacks sehr willkommen

Schlussfolgerungen

- Nebenläufigkeitsfehler sind heimtückisch
  - Nicht-deterministisches Auftreten
- Mix an Gegenmassnahmen ist geboten
  - Architektur, Testing, Analyse-Tools
- Neues Analyse-Tool «HSR Parallel Checker»
  - Erkennung von Data Races & Deadlocks in C#
  - Fokus: Schnell und möglichst exakt

Danke für Ihre Aufmerksamkeit

- Kontakt
  - **Prof. Dr. Luc Bläser**
    HSR Hochschule für Technik Rapperswil
    lblaeser@hsr.ch
  - **Parallel Checker**
    - http://parallel-checker.com
  - **HSR Concurrency Lab**
    - http://concurrency.ch
  - **Microsoft Innovation Center Rapperswil**
    - http://msic.ch