

# Checking Non-Deterministic Behavior in Unit Tests

Luc Bläser

HSR Hochschule für Technik Rapperswil  
Switzerland

# Concurrency is Omnipresent

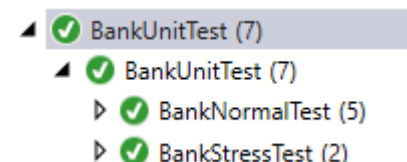
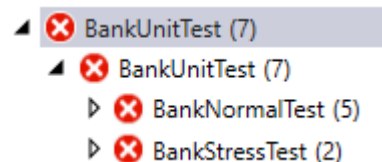
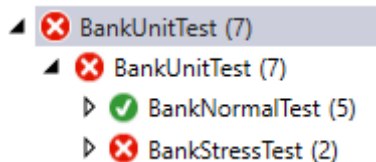
---

- Software becomes increasingly concurrent
- Mostly through implicit multi-threading
  - Asynchronous or reactive programming,
  - Task parallelization, thread pools
  - Parallel querying, parallel loops
  - Timers, finalizers etc.

# Challenge for Testing

---

- Non-deterministic execution
  - Issues may only occur in certain schedules
  - Bugs can appear seldom or never in tests
- Consequences
  - Tests sometimes green, sometimes red
  - Unreliable regression testing



# Types of Concurrency Errors

---

- **Race conditions**
  - Errors because of insufficient synchronization
- **Data races**
  - Unsynchronized concurrent RW or WW accesses to same variable or array element
- **Deadlocks**
  - Cyclic lock-and-wait dependencies
- **Livelocks**
  - Cyclic perpetual wait dependencies (spinning threads)
- **Starvation**
  - Continuous progress prevention (with chance to recover)

Requires knowledge of program semantics (intended behavior, progress)

# Frequent Practitioner's Approach

---

## Concurrent stress tests

- Many threads call operations
- Check assertions at the end
- Insert extra synchronization in tests

- + No extra tools
- Sporadic occurrence
- Not reproducible
- Code effort

# Concurrent Unit Test

---

```
[TestMethod]
[Timeout(TestTimeout)]
public void TestConcurrentDeposits() {
    const int N = 10;
    var account = new BankAccount();
    var list = new List<Thread>();
    for (int count = 0; count < N; count++) {
        var thread = new Thread(() => account.Deposit(1));
        thread.Start();
        list.Add(thread);
    }
    foreach (var thread in list) {
        thread.Join();
    }
    Assert.AreEqual(N, account.Balance);
}
```

Timeout in case of  
deadlocks/blocking

Join before assertion

Check final balance  
(race condition)

# Systematic Approaches

---

- Dynamic analysis (e.g. ThreadSanitizer, Inspector)
  - + Precise
  - Sporadic occurrence
  - Not reproducible
- Static analysis (e.g. CHES, JPF)
  - + Completeness
  - False positives
  - State explosion
- Sound + precise in general = halting problem

# Hybrid Approaches

---

- Dynamic + static
  - Run, analyze trace, statically derive alternative traces
  - E.g. Concolic Testing, Predictive Testing
  - + Precision
  - Expensive constraint solver
- Systematic concrete interpretation
  - Exhaustive testing towards full schedule coverage
  - E.g. CHESSE, JPF
  - + Precision
  - State explosion



# Goals for Our Checker

---

- **Extensive:** Analyze many schedules (but not all)
- **Fast:** Few seconds even for large code
- **Reproducible:** Always report the same issues
- **Precise:** Few false warnings

But not complete, may miss issues

Initially designed for use in an IDE

Question: Could it be used for testing as well?

# ISSTA 2018 Paper of Checker

---



## Practical Detection of Concurrency Issues at Coding Time

Luc Bläser

HSR Hochschule für Technik Rapperswil

Rapperswil, Switzerland

lblaeser@hsr.ch

### ABSTRACT

We have developed a practical static checker that is designed to interactively mark data races and deadlocks in program source code at development time. As this use case requires a checker to be both fast and precise, we engaged a simple technique of randomized bounded concrete concurrent interpretation that is experimentally effective for this purpose. Implemented as a tool for C# in Visual Studio, the checker covers the broad spectrum of concurrent language concepts, including task and data parallelism, asynchronous programming, UI dispatching, the various synchronization primitives, monitor, atomic and volatile accesses, and finalizers. Its application to popular open-source C# projects revealed several real issues with only a few false positives.

### CCS CONCEPTS

• **Software and its engineering** → **Concurrent programming structures; Software defect analysis;**

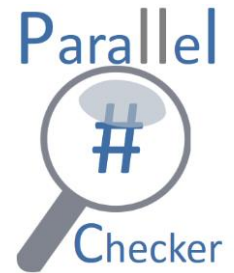
current language versions. Static concurrency analysis continues to be an area of research where very few practical tools [26, 36] are on hand. For newer C# versions, there even exists no static checker for data races or deadlocks at all. Previous tools such as CHESS [24] have been discontinued. The situation is discussed in more detail in Section 6.

In this work, we aim to provide a practical tool that detects common concurrency errors in a slightly different setting than other work in this area. This tool should interactively support software developers when working in an integrated development environment (IDE): It should directly highlight problematic program sections with regard to concurrency during the coding. For this purpose, the following checker properties were considered essential:

- **Static:** The source code as displayed in the IDE needs to be analyzed. The code being written can be incomplete or contain erroneous fragments, making a program execution and thus a dynamic analysis impossible.

# HSR Parallel Checker

---

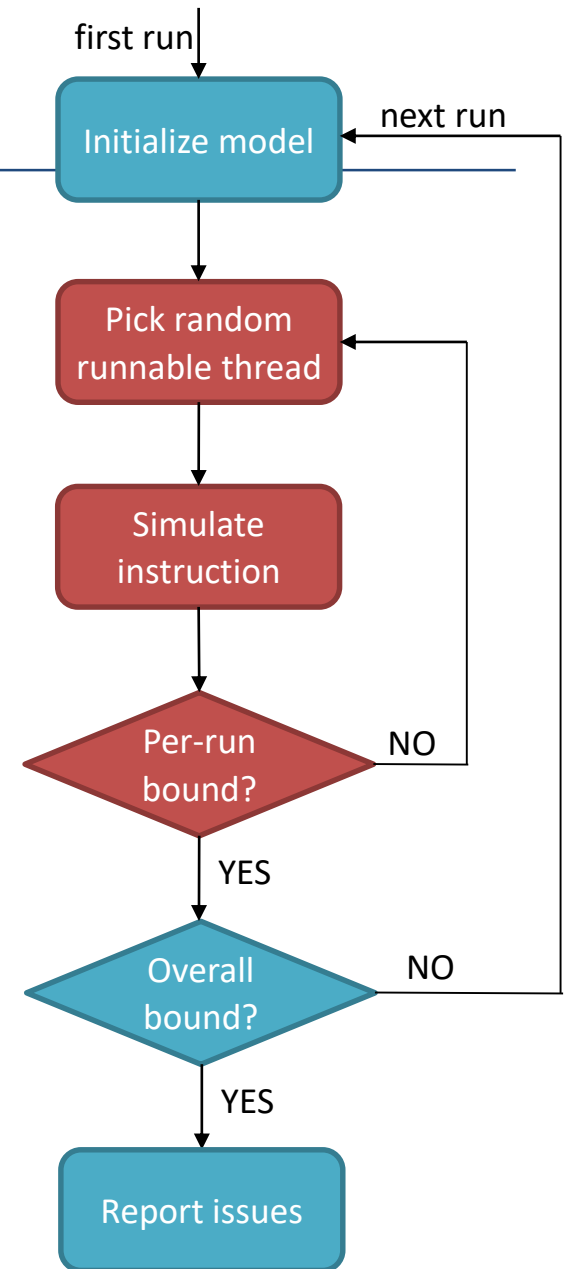


- Static checker tool for Visual Studio IDE
- For C#, covering wide concurrency spectrum
  - Tasks, async/await, parallel loops, various sync. constructs, atomics, volatile, finalizers, timers, parallel queries ...
  - UI-apps/libraries/**unit tests**/console-apps
- Downloadable on Visual Studio Marketplace (>2.5k installs)

# Approach

Randomized mostly-concrete interpretation

- Map code to internal runtime model
- Simulate execution on this model
- Maintain exact state where possible
- Repeated random scheduling
- Per-run and overall bound
- Report encountered issues
- Vector clock for race detection



# Particular Aspects

---

- Reproducibility of results
  - Seeded pseudo-random numbers
  - Bounds on logical number of steps and size
- Dynamic technique in static context
  - Does not run the code
  - Code may be incomplete or incorrect
- Deliberately simple design
  - Random scheduling, no constraint solver
  - Examine more code with less sophistication

# Abstract States

---

- Cope with unknown external input
  - Uninterpreted value
  - Imprecise assumptions (branches, locks, thread starts etc.)
  - May result in false positives (and false negatives)
- Today's focus: Unit Tests
  - Full input should be defined, no user interaction
  - Checkers becomes entirely concrete
  - No false positives

# IDE Checker Demo

The screenshot displays the Microsoft Visual Studio IDE interface. The main window shows the source code for `BankAccount.cs` with the following methods:

```
public void Deposit(int amount)
{
    lock (_sync)
    {
        balance += amount;
    }
}

public bool Withdraw(int amount)
{
    lock (_sync)
    {
        if (_balance >= amount)
        {
            balance -= amount;
            return true;
        }
        return false;
    }
}

public int Balance => balance;
```

The `Withdraw` method is highlighted. The `Balance` property is also highlighted. The `lock (_sync)` blocks are visible in both methods.

The **Error List** at the bottom shows the following error:

Code	Description	Project	File	Line	Su...
ParallelChecker	Detection in 332 ms (2 issues) Issue: #0 Data race on BankTest.BankAccount._balance caused by write at "_balance += amount" in BankAccount.cs line 12 caused by thread or task at "()" => { account.Deposit(100); var result = account.Withdraw(100); Console.WriteLine(result); }" in BankTest.cs line 11 caused by main thread at "Main" in BankTest.cs line 8 caused by read at "_balance" in BankAccount.cs line 29 caused by main thread at "Main" in BankTest.cs line 8 Issue: #1 Data race on BankTest.BankAccount._balance	BankTest		1	Active
ParallelChecker		BankTest	BankAccount.cs	12	Active

# Application to Testing

---

- Run checker by opening unit test source
  - Checker uses unit test method as entries
  - See errors in source code in IDE
- Run checker inside unit test framework
  - Run each unit test through the checker
  - See green/red unit test result



# Parallel Unit Test Demo

The screenshot displays the Microsoft Visual Studio interface during a unit test execution. The main window shows the source code for `BankNormalTest.cs` with the following content:

```
using System.Collections.Generic;
using System.Threading;
using BankTest;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace BankUnitTest {
    [TestClass]
    public class BankStressTest {
        [TestMethod]
        public void ConcurrentDepositStressTestNoRace() {
            const int N = 10, K = 10;
            var account = new BankAccount();
            var list = new List<Thread>();
            for (int count = 0; count < N; count++) {
                var thread = new Thread(() => {
                    for (int run = 0; run < K; run++) {
                        account.Deposit(1);
                    }
                });
            }
        }
    }
}
```

The Test Explorer on the left shows the test results for `BankUnitTest`. The `BankNormalTest` class has 5 tests, with `DepositRace` failing. The Error List at the bottom shows the following error:

Code	Description	Project	File	Line	S.
ParallelCI	Detection in 2857 ms (8 issues) Issue: #0 Data race on BankTest.BankAccount.Balance caused by write at "Balance += amount" in BankAccount.cs line 7 caused by thread or task at "() => account.Deposit(100)"	BankUnitTest		1	Active
ParallelCI	in BankNormalTest.cs line 11 caused by unit test thread caused by read at "account.Balance" in BankNormalTest.cs line 12 caused by unit test thread Issue: #0 Data race on BankTest.BankAccount.Balance caused by write at "Balance += amount" in BankAccount.cs line 7 caused by thread or task at "() => account.Deposit(100)"	BankUnitTest	BankNormalTest...	12	Active
ParallelCI	in BankNormalTest.cs line 11	BankUnitTest	BankAccount.cs	7	Active

The Properties window shows the source of the error: `BankNormalTest.cs` line 9. The message indicates a data race on `BankTest.BankAccount.Balance` caused by write and read operations in different threads.

# Conclusion

---

- Testing is difficult – in particular for concurrency
  - Non-deterministic bug occurrence
  - Hard to reproduce, hard to detect at all
- Dynamic testing within static analysis
  - Our checker does this to be precise and reproducible
- Static analysis within dynamic testing
  - Unit tests could again run through the static checker

# Thank You for Your Attention!

---

## ■ Contact

- Luc Bläser, HSR Hochschule für Technik Rapperswil
- [lblaeser@hsr.ch](mailto:lblaeser@hsr.ch), <http://concurrency.ch>

## ■ Project Website

- <http://parallel-checker.com>

## ■ VS Marketplace

- <https://marketplace.visualstudio.com/items?itemName=L.BHSR.HSRParallelCheckerforC7VS2017>